ELEE 4220/6220: Feedback Controls

Lab 4 – Spring 2025 – Emerson Hall

Lab 2: Motor Control

Deliverables:

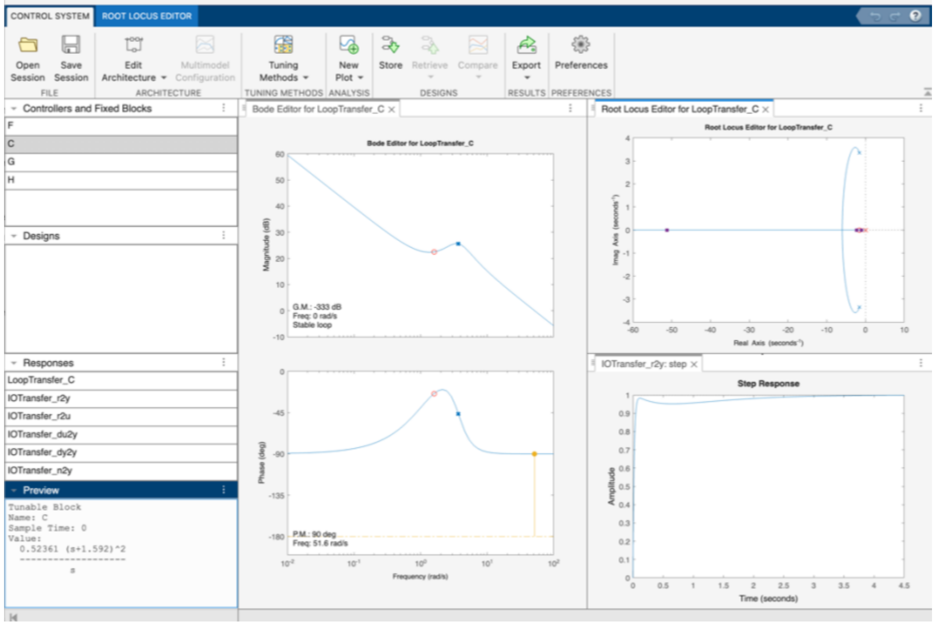
1. **Describe what method you are implementing and show your first order transfer function model.**
    - **a.** Considering the actuation method for controlling the DC motor, I have chosen to represent the controller output as a duty cycle, limiting it to a range of [0,1]. This approach aligns with the operational characteristics of the SSR, which modulates power delivery through Pulse Width Modulation (PWM). I actually created a new transfer function of
      $P(s) = \frac{98.22}{s^2+3.17s+13.76}$
      where:
      98.22 represents the system gain,
      $s^2 + 3.17s + 13.76$ models the motor's dynamics, incorporating the effects of inertia and damping. The term 3.17s accounts for the system's damping, while 13.76 represents the squared natural frequency of the motor system.
    - **b.** The chosen duty cycle-based control ensures smooth speed regulation, optimizing the response time and stability while staying within the physical limitations of the system. This decision lays the groundwork for implementing a PID controller to refine performance, ensuring minimal steady-state error and an appropriate transient response, achieving a system that is close to critically damped.

2. **From the reading in Chapter 1 of Practical PID, different types of PID controllers can be implemented.**
    - **a. What are the pros/cons of on/off control?**
        - **i.** The pros of on/off controls are that it is simple to implement, there's no need for tuning, works well for basic systems and it is cost-effective.
        - **ii.** The cons are that it causes oscillations, there's limited precision, hysteresis is required to prevent excessive switching, and not suitable for smooth control.
    - **b. What are the different topologies of PID control?**
        - **i.** Ideal (Non-Interacting) PID
            1. $C_i(s) = K_p(1 + \frac{1}{T_i s} + T_d s$
            2. Standard form, easy to understand
            3. Each term operates independently
        - **ii.** Series (interacting) PID
            1. $C_s(s) = K_p'(1 + \frac{1}{T_i' s})(T_d' s + 1)$
            2. Historically used in pneumatic controllers
            3. More challenging to tune than the ideal form
        - **iii.** Parallel PID (Independent Gains)
            1. $C_p(s) = K_p + \frac{K_i}{s} + K_d s$
            2. Each gain is fully independent

            3. Used in MATLAB's pid() function
        iv. Velocity PID (Incremental):
            1. Computes change in control signal rather than an absolute value
            2. Used in embedded systems where real-time updates are needed
    c. **Which PID topology are you implementing for this lab and explain why?**
        i. For this lab, I am using the Parallel PID controller. I will be able to use MATLAB and the parallel PID is easier to tune. Also, each term is controlled separately.

3. **From the reading in Chapter 2 of Practical PID, noise corrupts the intent of the derivative term in the PID controller. One solution is to use "N".**
    a. **Explain what "N" is.**
        i. "N" is the derivative filter constant, used to limit the amplification of the high-frequency noise in the derivative term. It is needed as it magnifies high-frequency noise in sensor data and without N, the actuator may receive excessive, erratic commands.
    b. **Explain the range of "N".**
        i. The typical range of N is between 5 and 20. Low N values provides strong filtering but reduces the effectiveness of the derivative action. The system might respond slower to changes. With high N values, the derivative term remains almost unchanged. The system suffers from excessive noise sensitivity.
    c. **Return to looking at Figure 1. If the filtered data is going to our controller, what should be updated about the controller gains?**
        i. The gains should be recalibrated when using the filter data. Proportional gain (Kp) should be slightly increased since high-frequency noise is attenuated. Integral gain (Ki) might need a small increase to compensate for filtering effects but the integral action remains mostly unchanged. The derivative gain (Kd) can be increased slightly with a higher N. The choice of "N" affects how much we can increase Kd. If N is too small, higher Kd is needed. If N is between 8-16 (practical range), Kd can be moderately increased. If N is too high, instability can happen if Kd is too high.

4. **Use Control system designer in MATLAB to design a PID controller to track velocity of the motor.**

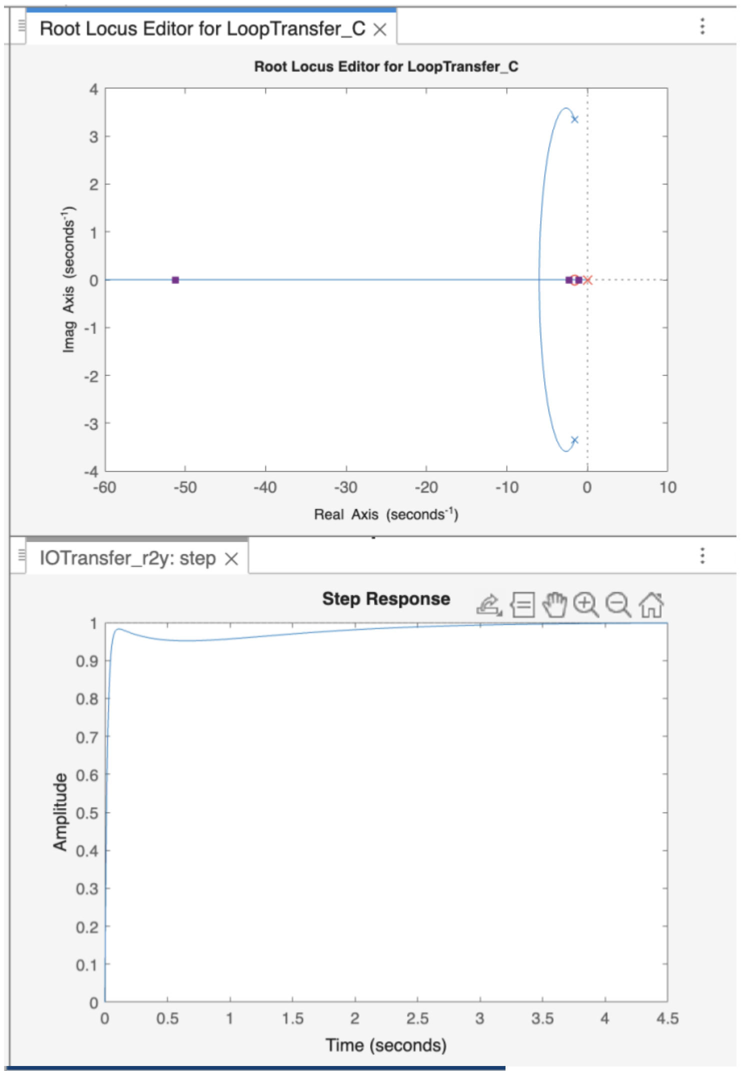


```
Tunable Block
Name: C
Sample Time: 0
Value:
  0.52361 (s+1.592)^2
  -------------------
           s
```

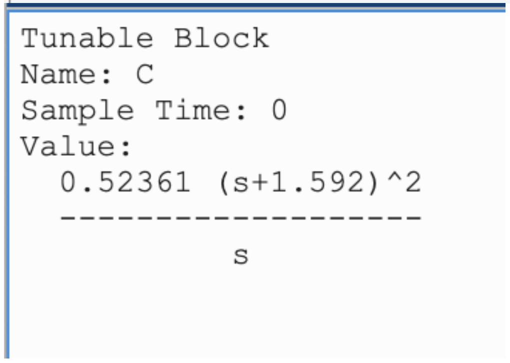
a. **The system should have zero steady-state error. The system should be close to critically damped. Here close is used since this is rarely ever perfectly achieved.**

b. **Include screenshots of your pole/zero map, final controller, and write justifications for the controller.**

   i. The Parallel PID I made has Kp = 1.665, Ki = 1.328, Kd = 0.52361.



ii.

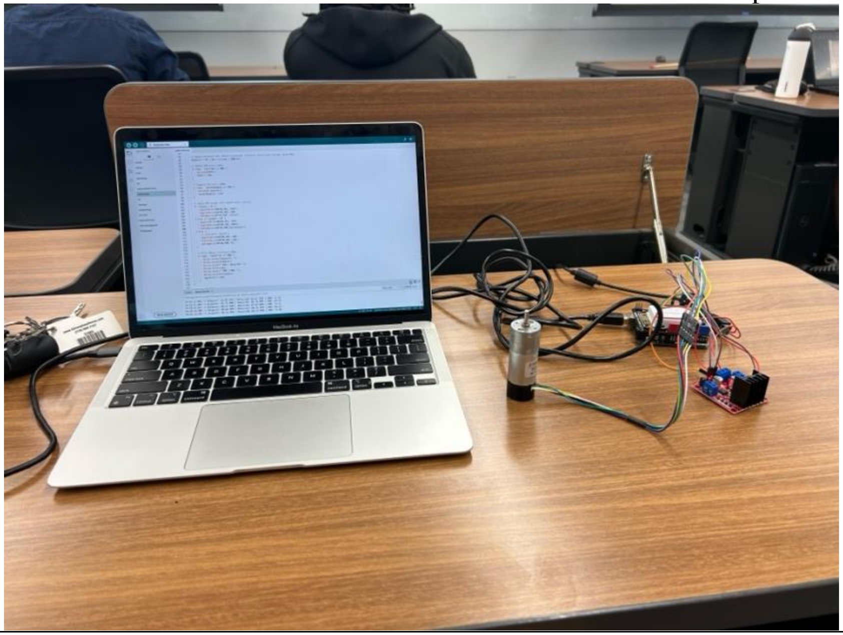


```
Tunable Block
Name: C
Sample Time: 0
Value:
  0.52361 (s+1.592)^2
  -------------------
           s
```

iii.

iv. This Parallel PID Controller was selected to achieve a balance between response speed, stability, and accuracy. The low gains prevent excessive oscillations, while the integral action ensures proper tracking of the reference velocity. The design keeps noise amplification minimal and provides a robust control solution for the DC motor system.

5. **Implement the PID controller to track an interesting reference velocity trajectory. An interesting trajectory is defined as: 1) is within range of the motors speed and 2) is more than just a bump test.**
   a. **Include screenshots of the software and hardware setup.**
      i. I had to use Simulink Control System Designer to make the PID controller but I had to use Arduino IDE since Simulink does not sample fast enough.



      ii.

```
#include <PID_v1.h>


// Motor Control Pins
#define MOTOR_PWM 9  // PWM output (ENA - controls speed)
#define MOTOR_IN1 5  // Motor Direction (IN1)
#define MOTOR_IN2 6  // Motor Direction (IN2)


// Encoder Pins
#define ENCODER_A 2  // Encoder Signal A
#define ENCODER_B 3  // Encoder Signal B


// PID Variables
double setpoint = 50;  // Desired RPM
double input, output;
double Kp = 1.665, Ki = 1.328, Kd = 0.52361;  // PID tuning parameters


// Create PID Controller
PID motorPID(&input, &output, &setpoint, Kp, Ki, Kd, DIRECT);
```

```
// Encoder Variables
volatile int encoderCount = 0;
unsigned long lastTime = 0;
double rpm = 0;
const int PPR = 540;  // Pulses per revolution for your encoder

// Moving Average Buffer for RPM Smoothing
double rpmBuffer[5] = {0};
int bufferIndex = 0;

// Interrupt Service Routine for Encoder
void encoderISR() {
   encoderCount++;  // Increment count every time encoder A changes
}

// Function to Calculate RPM with Moving Average Filtering
void calculateRPM() {
   unsigned long currentTime = millis();
   double elapsedTime = (currentTime - lastTime) / 1000.0;  // Convert ms to seconds

   if (elapsedTime > 0 && encoderCount > 0) {
      double newRPM = (encoderCount * 60.0) / (PPR * elapsedTime);  // Corrected PPR = 540

      // Apply Moving Average Filtering
      rpmBuffer[bufferIndex] = newRPM;
      bufferIndex = (bufferIndex + 1) % 5;  // Keep buffer index in range

      double sum = 0;
      for (int i = 0; i < 5; i++) sum += rpmBuffer[i];
      rpm = sum / 5;  // Compute average
   } else {
      rpm = 0;
   }

   encoderCount = 0;  // Reset count after reading
   lastTime = currentTime;
```

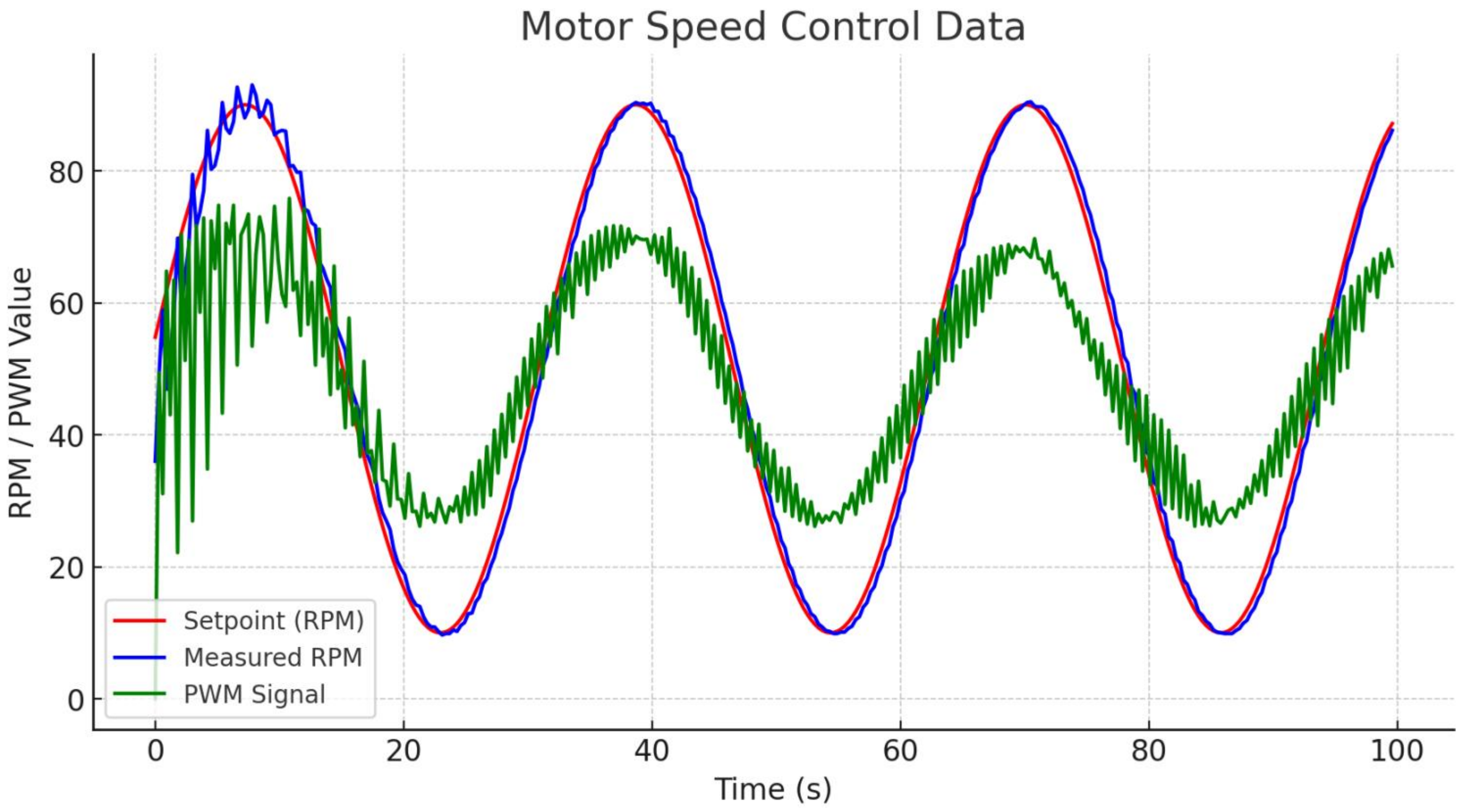
```
}

void setup() {
  Serial.begin(2000000);

  // Setup Encoder Interrupt
  pinMode(ENCODER_A, INPUT);
  pinMode(ENCODER_B, INPUT);
  attachInterrupt(digitalPinToInterrupt(ENCODER_A), encoderISR, CHANGE);

  // Setup Motor Control
  pinMode(MOTOR_PWM, OUTPUT);
  pinMode(MOTOR_IN1, OUTPUT);
  pinMode(MOTOR_IN2, OUTPUT);
  digitalWrite(MOTOR_IN1, LOW);
  digitalWrite(MOTOR_IN2, LOW);

  // Initialize PID
  motorPID.SetMode(AUTOMATIC);
  motorPID.SetOutputLimits(0, 255);  // PWM range
}

// Variables to control update timing
unsigned long lastPIDUpdate = 0;
unsigned long lastPrint = 0;

void loop() {
  unsigned long now = millis();

  // Update setpoint with smooth sinusoidal reference (oscillates between 10-90 RPM)
  setpoint = 50 + 40 * sin(now / 5000.0);

  // Update RPM every 200ms
  if (now - lastTime >= 200) {
    calculateRPM();
    input = rpm;
  }
```

```
    // Compute PID every 100ms
    if (now - lastPIDUpdate >= 100) {
        motorPID.Compute();
        lastPIDUpdate = now;
    }

    // Apply PID output with smooth motor control
    if (output > 0) {
        digitalWrite(MOTOR_IN1, HIGH);
        digitalWrite(MOTOR_IN2, LOW);
        analogWrite(MOTOR_PWM, output);
    } else if (output < 0) {
        digitalWrite(MOTOR_IN1, LOW);
        digitalWrite(MOTOR_IN2, HIGH);
        analogWrite(MOTOR_PWM, abs(output));
    } else {
        // Stop motor smoothly
        digitalWrite(MOTOR_IN1, LOW);
        digitalWrite(MOTOR_IN2, LOW);
        analogWrite(MOTOR_PWM, 0);
    }

    // Print Debug Info Every 300ms
    if (now - lastPrint >= 300) {
        Serial.print("Setpoint: ");
        Serial.print(setpoint);
        Serial.print(" RPM | Measured: ");
        Serial.print(rpm);
        Serial.print(" RPM | PWM: ");
        Serial.println(output);
        lastPrint = now;
    }
}
```

6.

7.

a. **Include a plot of the desired trajectory and the closed-loop response.**

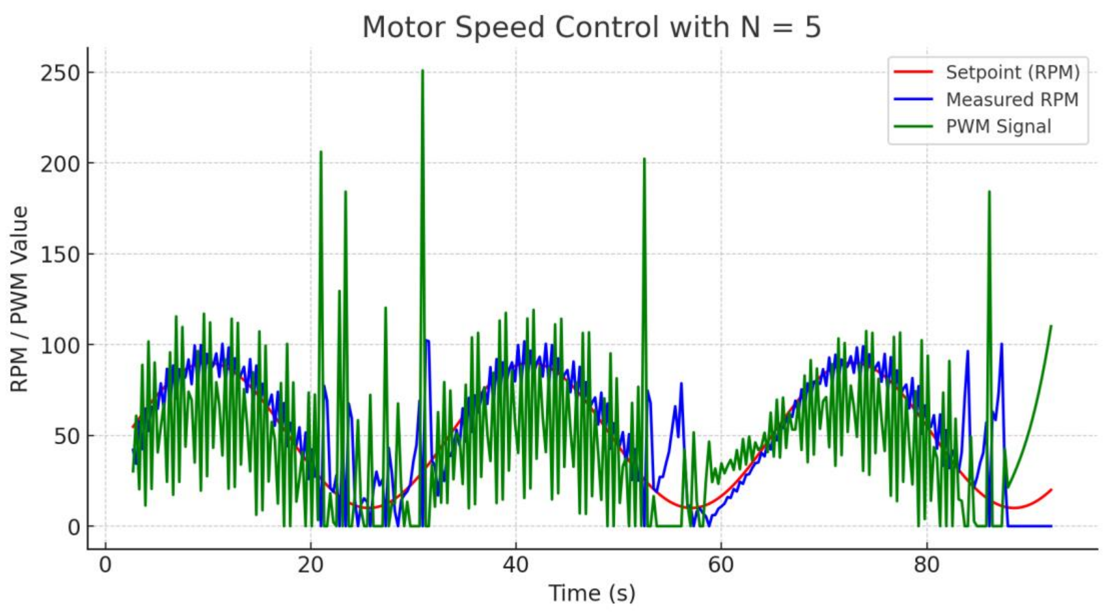


b. **Include an analysis of the results.**

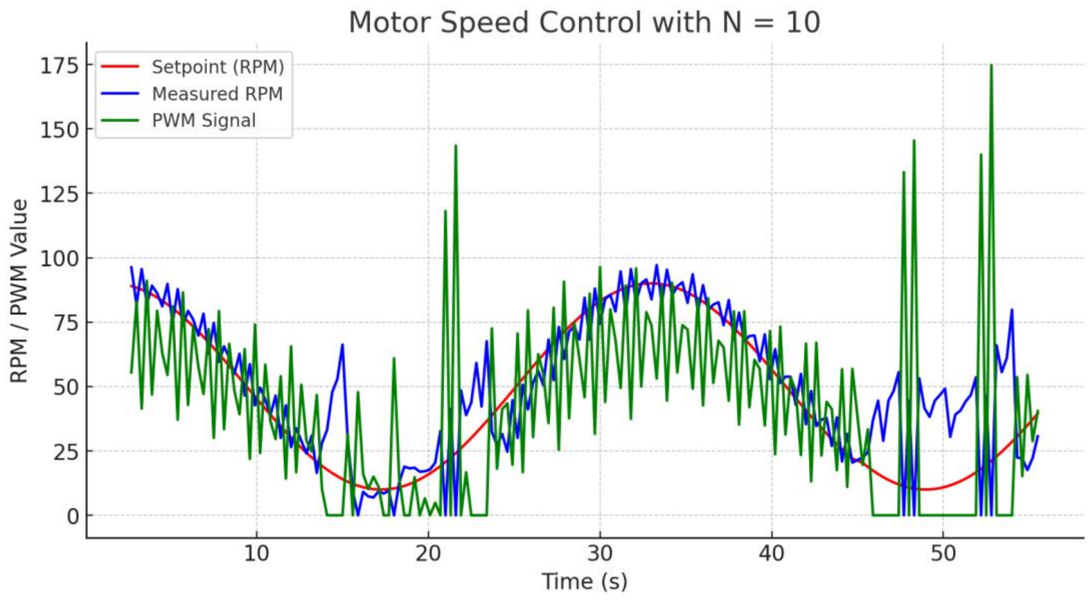
   i. The results indicate that the PID controller effectively regulates motor speed, as the measured RPM closely follows the setpoint over time. Initially, there are small deviations where the measured RPM lags behind the setpoint, suggesting a slight delay in the motor's response to changes. However, as the system progresses, the steady-state error decreases, and the measured RPM stabilizes near the setpoint, demonstrating good tracking performance. The PWM signal adjusts dynamically to compensate for errors, showing higher values when the motor needs to accelerate and lower values when corrections are needed. Some fluctuations in measured RPM suggest that minor tuning adjustments, such as increasing the proportional gain (Kp) or fine-tuning the derivative gain (Kd), could enhance responsiveness and reduce tracking delays. Overall, the controller performs well, maintaining accurate speed control with minimal steady-state error.

8. **Return to the discussion with "N" earlier. Repeat the test with various values of "N".**

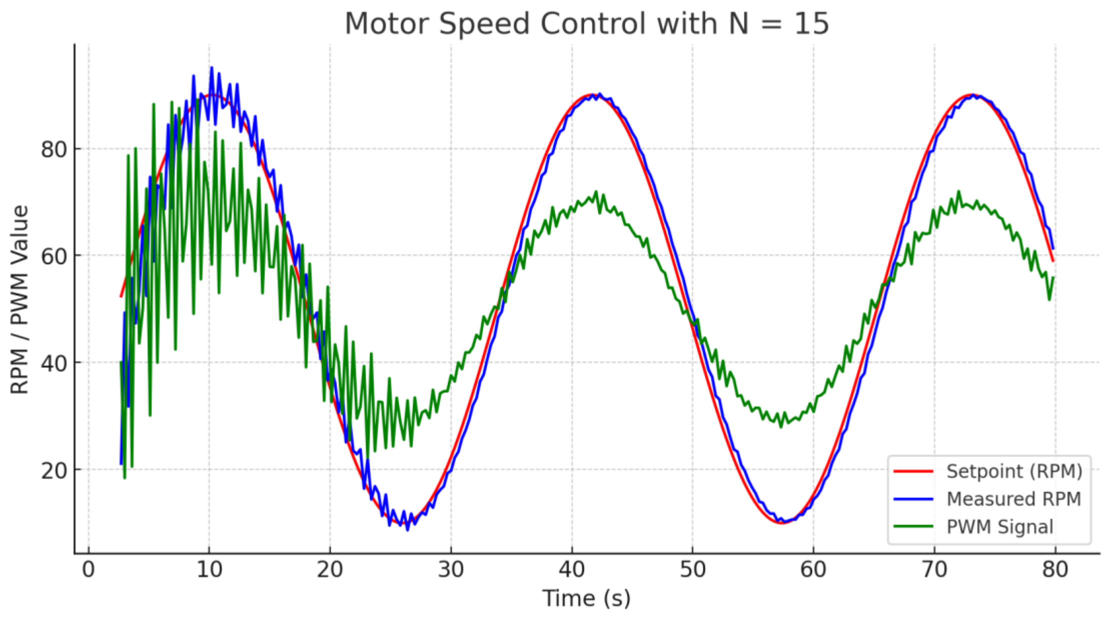
   a. **Include trajectory comparisons.**

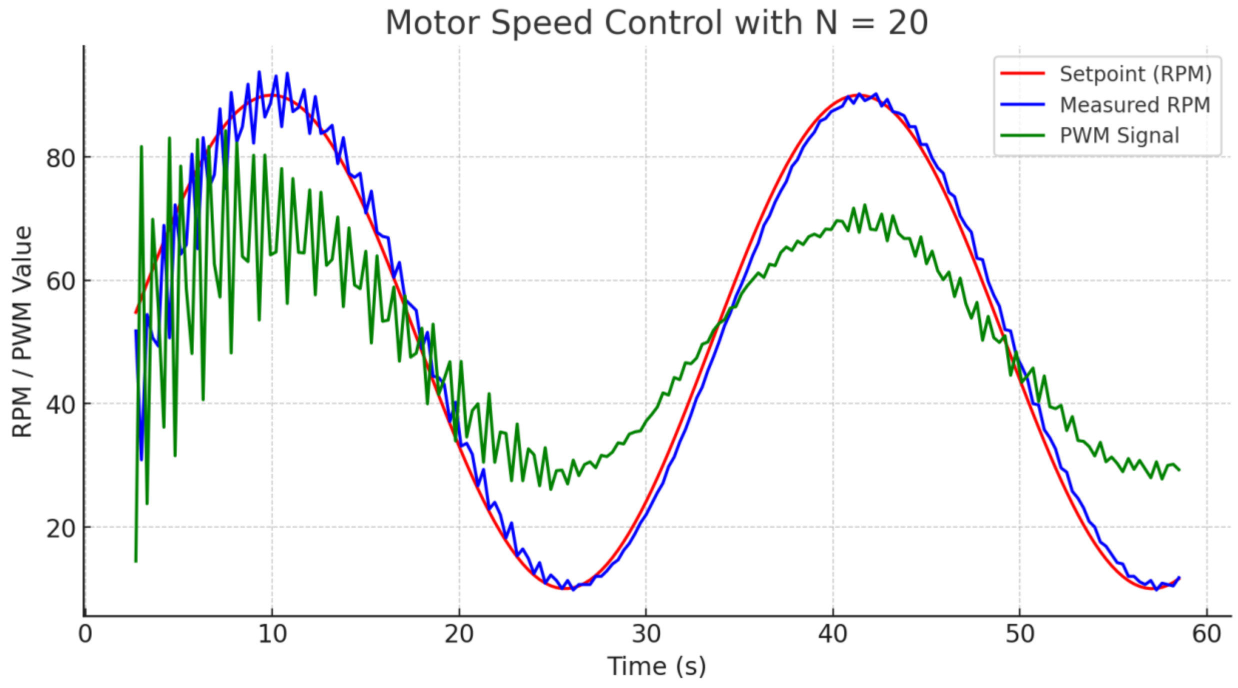


Motor Speed Control with N = 5
Setpoint (RPM)
Measured RPM
PWM Signal
RPM / PWM Value
Time (s)
0
50
100
150
200
250
0
20
40
60
80



Motor Speed Control with N = 10
Setpoint (RPM)
Measured RPM
PWM Signal
RPM / PWM Value
Time (s)
0
25
50
75
100
125
150
175
10
20
30
40
50





b. Include analysis of changing "N".

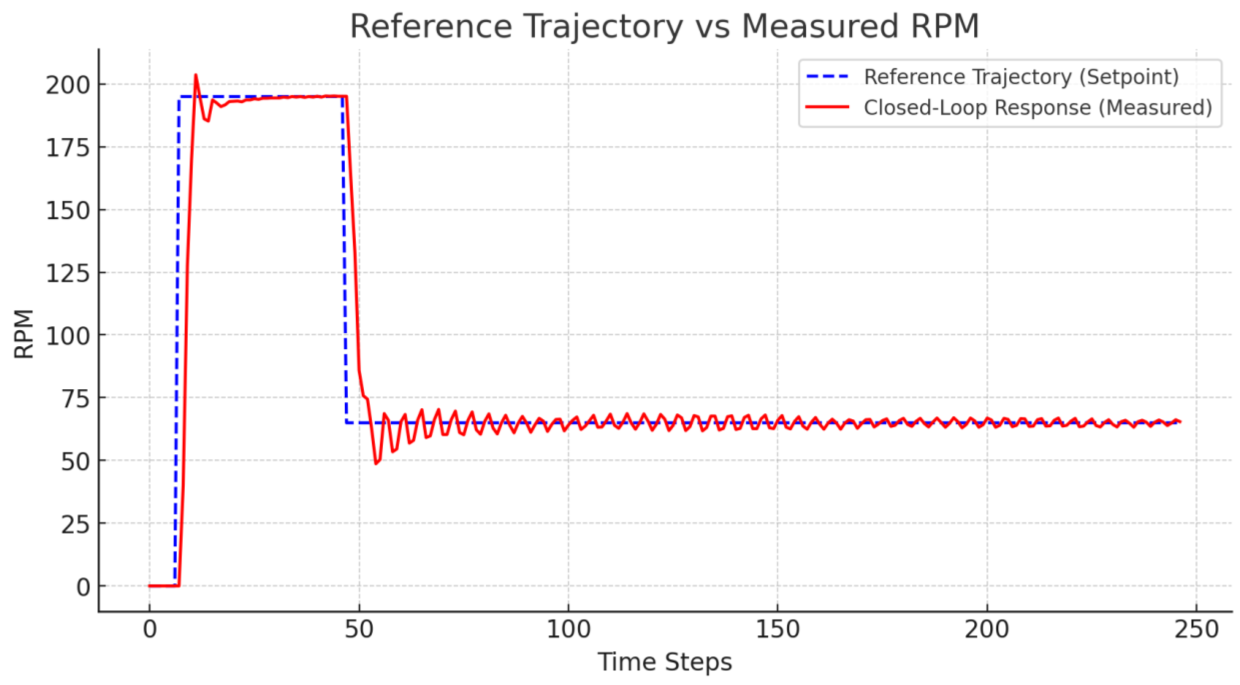
i. With N = 5, the system exhibited very slow response, with the measured RPM appearing almost static, failing to react quickly to changes in the setpoint. Additionally, the PWM signal fluctuated erratically, likely due to excessive filtering weakening the derivative term's influence, causing sluggish adjustments. In contrast, N = 15 and N = 20 showed much better performance, with initial instability in both measured RPM and PWM but eventually settling and closely matching the setpoint. The higher derivative action at these values allowed the system to react more effectively, reducing lag and improving tracking. While at first, the measured RPM was sporadic, it gradually stabilized, indicating that the controller was effectively adapting. The PWM signal, though chaotic at the beginning, became more consistent over time, showing that the system was tuning itself towards a steady state. Overall, N = 15 and N = 20 provided the best balance between responsiveness and stability, while N = 5 severely limited system performance due to excessive filtering.

9. **Now we move to Chapter 3 of Practical PID. Create a new semi-interesting reference velocity trajectory. Step the system to a final value of 1.5x the maximum speed of the motor and wait *awhile*. Here *awhile* is defined as ~3x the projected steady-state value (projected since the system will never reach the final value). Next, have the trajectory step down to 0.5x the maximum speed of the motor. Have the system run until steady-state is reached.**

```
#include <PID_v1.h>


// Motor Control Pins
#define MOTOR_PWM 9  // PWM output (ENA - controls speed)
#define MOTOR_IN1 5  // Motor Direction (IN1)
#define MOTOR_IN2 6  // Motor Direction (IN2)


// Encoder Pins
#define ENCODER_A 2  // Encoder Signal A
#define ENCODER_B 3  // Encoder Signal B


// PID Variables
double setpoint = 0;  // Start at 0 RPM
double input, output;
double Kp = 1.665, Ki = 1.328, Kd = 0.52361;  // PID tuning parameters


// Motor Specifications
const double MAX_SPEED = 130;  // Max speed in RPM (from motor specs)
double high_speed = 1.5 * MAX_SPEED;  // Step to 1.5x max speed
double low_speed = 0.5 * MAX_SPEED;   // Step to 0.5x max speed


// Create PID Controller
PID motorPID(&input, &output, &setpoint, Kp, Ki, Kd, DIRECT);


// Encoder Variables
volatile int encoderCount = 0;
```

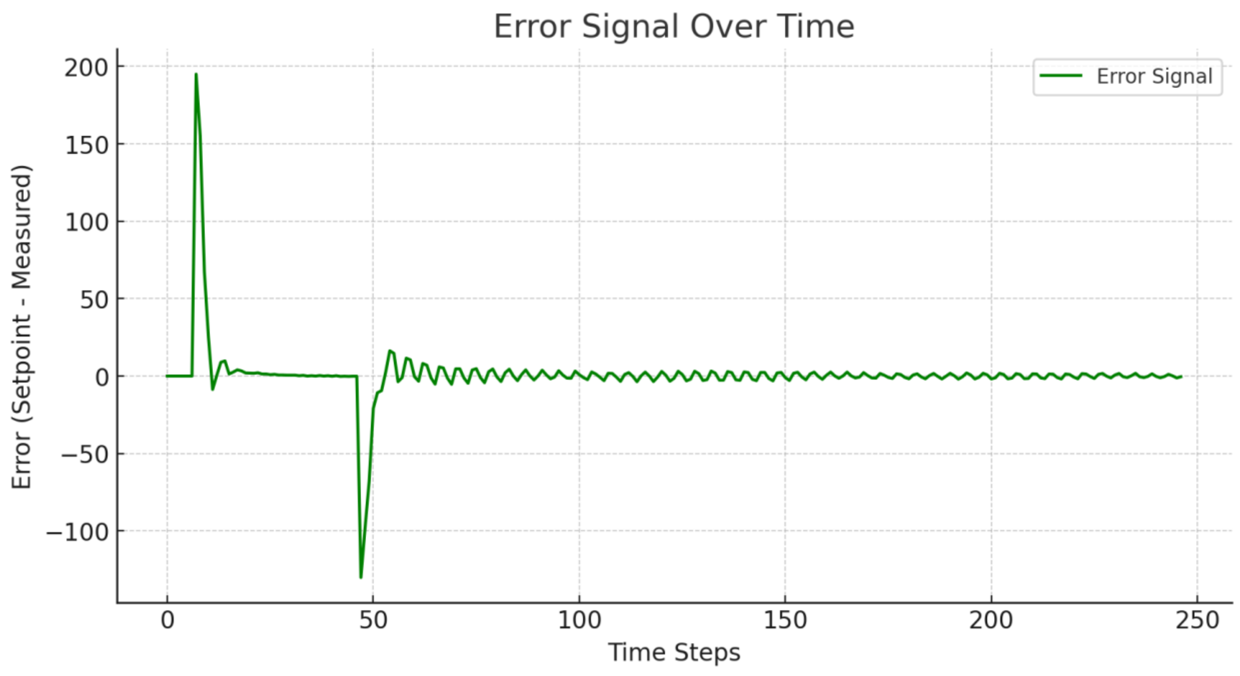
```
unsigned long lastTime = 0;
double rpm = 0;
const int PPR = 540;  // Pulses per revolution for your encoder

// Moving Average Buffer for RPM Smoothing
double rpmBuffer[5] = {0};
int bufferIndex = 0;

// Timing Variables for Step Trajectory
unsigned long phaseStartTime = 0;
int phase = 0;  // Tracks current phase

// Debugging Variables
unsigned long lastPrint = 0;

// Interrupt Service Routine for Encoder
void encoderISR() {
  encoderCount++;
}

// Function to Calculate RPM with Moving Average Filtering
void calculateRPM() {
  unsigned long currentTime = millis();
  double elapsedTime = (currentTime - lastTime) / 1000.0;  // Convert ms to seconds

  if (elapsedTime > 0 && encoderCount > 0) {
    double newRPM = (encoderCount * 60.0) / (PPR * elapsedTime);

    // Apply Moving Average Filtering
    rpmBuffer[bufferIndex] = newRPM;
    bufferIndex = (bufferIndex + 1) % 5;

    double sum = 0;
    for (int i = 0; i < 5; i++) sum += rpmBuffer[i];
    rpm = sum / 5;
  } else {
    rpm = 0;
    }

    encoderCount = 0;
    lastTime = currentTime;
}

// Function to Update the Reference Trajectory
void updateSetpoint() {
    unsigned long now = millis();
    if (phase == 0) {  // Start at 0 RPM
        setpoint = 0;
        Serial.println("Phase 0: Motor stopped.");
        phaseStartTime = now;
        phase = 1;
    }
    else if (phase == 1 && now - phaseStartTime >= 2000) {  // Step to 1.5x max speed after 2s
        setpoint = high_speed;
        Serial.println("Phase 1: Increasing to 1.5x max speed.");
        phaseStartTime = now;
        phase = 2;
    }
    else if (phase == 2 && now - phaseStartTime >= 12000) {  // Hold for ~3x steady-state time
        setpoint = low_speed;
        Serial.println("Phase 2: Decreasing to 0.5x max speed.");
        phaseStartTime = now;
        phase = 3;
    }
    else if (phase == 3) {  // Hold at 0.5x max speed indefinitely
        setpoint = low_speed;
    }
}

void setup() {
    Serial.begin(2000000);
    delay(1000);

    // Setup Encoder Interrupt
    pinMode(ENCODER_A, INPUT);
    pinMode(ENCODER_B, INPUT);
    attachInterrupt(digitalPinToInterrupt(ENCODER_A), encoderISR, CHANGE);

    // Setup Motor Control
    pinMode(MOTOR_PWM, OUTPUT);
    pinMode(MOTOR_IN1, OUTPUT);
    pinMode(MOTOR_IN2, OUTPUT);
    digitalWrite(MOTOR_IN1, LOW);
    digitalWrite(MOTOR_IN2, LOW);

    // Initialize PID
    motorPID.SetMode(AUTOMATIC);
    motorPID.SetOutputLimits(0, 255);

    Serial.println("Initializing...");
    phaseStartTime = millis();  // Initialize time tracking
}

// Loop Timing Variables
unsigned long lastPIDUpdate = 0;

void loop() {
    unsigned long now = millis();

    // Update setpoint based on step reference trajectory
    updateSetpoint();

    // Update RPM every 200ms
    if (now - lastTime >= 200) {
        calculateRPM();
        input = rpm;
    }

    // Compute PID every 100ms
    if (now - lastPIDUpdate >= 100) {
        motorPID.Compute();
        lastPIDUpdate = now;
    }

    // Apply PID output
    if (output > 0) {
        digitalWrite(MOTOR_IN1, HIGH);
        digitalWrite(MOTOR_IN2, LOW);
        analogWrite(MOTOR_PWM, output);
    } else {
        digitalWrite(MOTOR_IN1, LOW);
        digitalWrite(MOTOR_IN2, LOW);
        analogWrite(MOTOR_PWM, 0);
    }

    // Print Debug Info Every 300ms
    if (now - lastPrint >= 300) {
        Serial.print("Phase: ");
        Serial.print(phase);
        Serial.print(" | Setpoint: ");
        Serial.print(setpoint);
        Serial.print(" RPM | Measured: ");
        Serial.print(rpm);
        Serial.print(" RPM | PWM: ");
        Serial.println(output);
        lastPrint = now;
    }
}
```

a. **Create a plot of the system.**
    i. **Show the reference trajectory vs the closed-loop response.**



b. Create a second plot of the system

i. Show the error signal

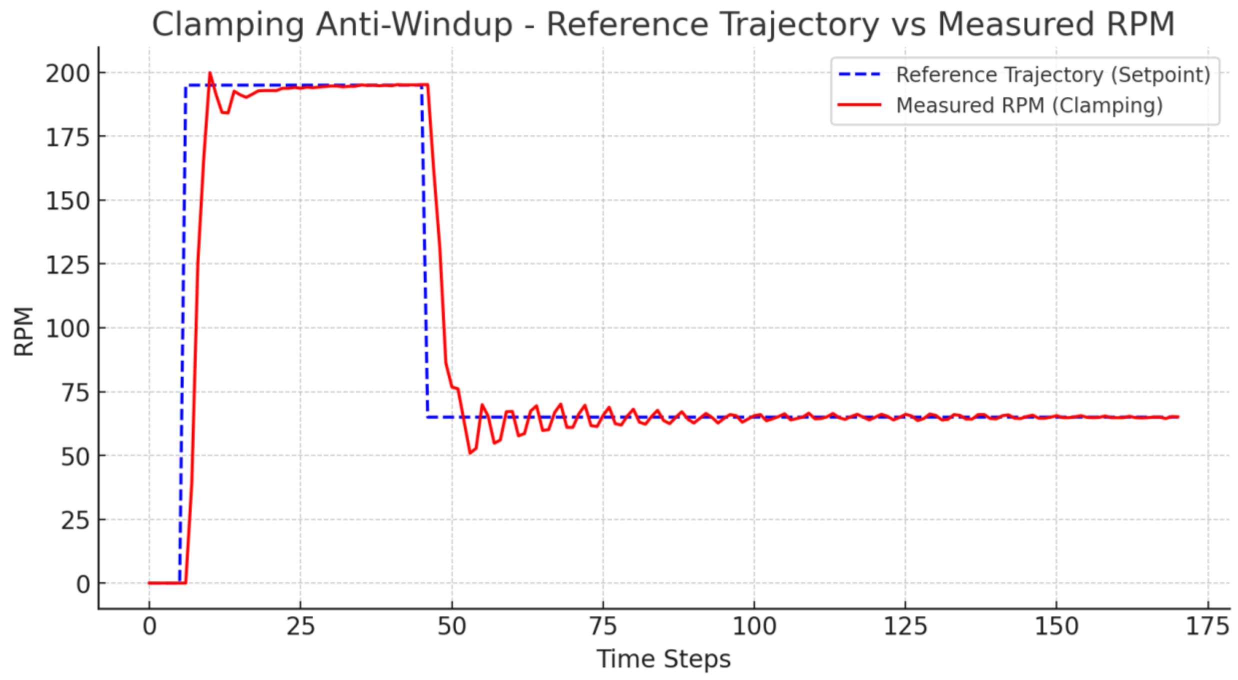


c. Explain why the system is "slow" to respond when the trajectory changes to the very realizable value of 0.5x max speed.

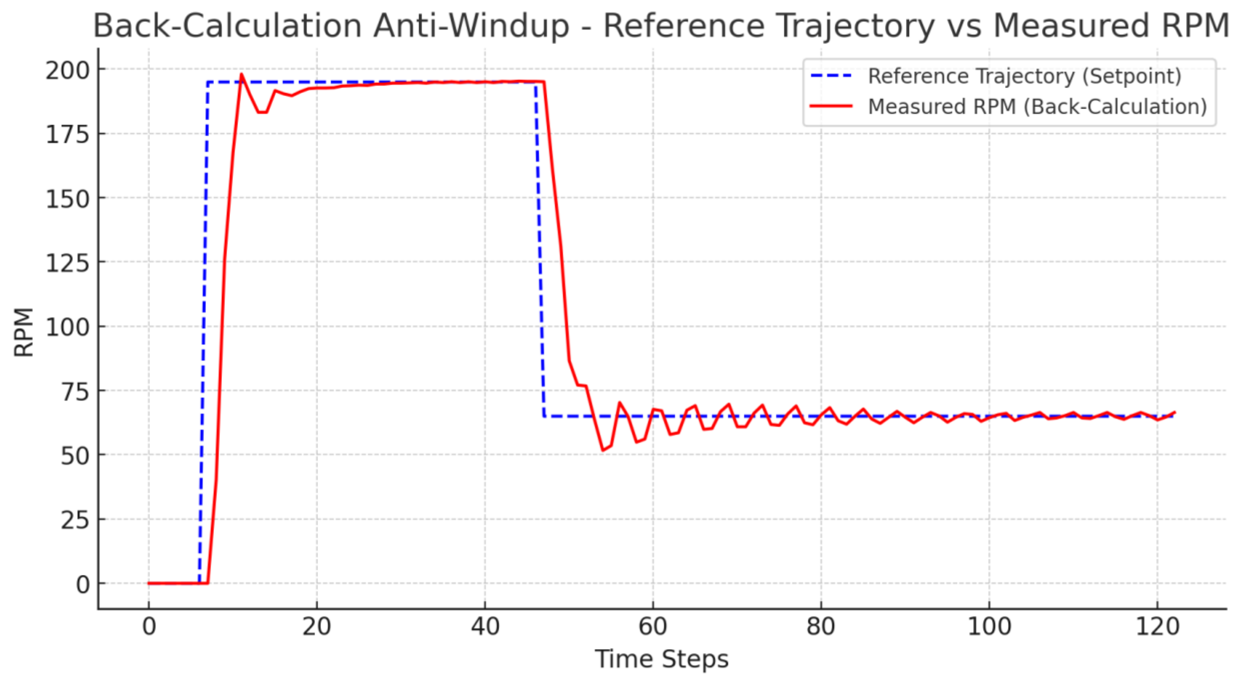
i. The system is slow to respond when the trajectory steps down to 0.5× max speed (65 RPM) due to a combination of mechanical inertia, PID tuning limitations, and motor dynamics. When operating at 1.5× max speed (195 RPM), the motor has significant rotational inertia, meaning it resists rapid deceleration. The PID controller must gradually reduce the PWM signal to slow the motor while avoiding excessive undershoot. However, at lower speeds, friction and load resistance have a greater proportional effect, requiring more precise control to maintain stability. Additionally, the derivative term (Kd) in the PID controller may not be sufficiently aggressive in damping the transition, leading to an extended settling time. If Ki (integral term) accumulates error from the high-speed phase, it can momentarily cause the system to resist the downward adjustment. These combined effects result in the system taking longer to stabilize at 65 RPM, making the response slower compared to an increase in speed.

10. Now use the MATLAB PID controller block and implement anti-windup strategies.

a. Document the addition of a clamping technique.



b. Document the addition of a back-calculation technique. Start with the gain suggestion given in Chapter 3 of Practical PID.



Gain was 1.328 same as Ki.

c. **Compare the results.**

i. The comparison between Clamping and Back-Calculation Anti-Windup reveals significant differences in control performance. The Clamping method effectively limits integral windup by preventing further accumulation when the output reaches saturation, but it introduces more oscillations and slower stabilization. This is evident in the measured RPM response, where Clamping exhibits greater overshoot and fluctuations before settling. In contrast, Back-Calculation Anti-Windup provides a smoother and more stable response by dynamically adjusting the integral term when saturation occurs. The measured RPM closely follows the setpoint, reducing overshoot and stabilizing faster than Clamping. The error signal further confirms this, with Back-Calculation displaying smaller error deviations and quicker convergence to zero, while Clamping shows larger fluctuations and slower correction. These results demonstrate that Back-Calculation is the superior anti-windup strategy, providing better tracking accuracy, reduced overshoot, and improved system stability compared to Clamping.

11. Explain how you would code:

a. **A clamping technique**

i. The Clamping Anti-Windup technique prevents excessive integral accumulation when the controller output reaches its saturation limits. This is achieved by freezing the integral term when the output is at its maximum or minimum. In code, this is implemented by checking if the computed PID output exceeds the actuator limits (e.g., 0 to 255 for PWM control). If saturation occurs, the integral component is disabled or held constant to prevent further accumulation. A simple way to implement this in a PID loop is by setting Ki = 0 when the output is saturated and restoring it when within bounds.

```
    if (output > 255) {
        output = 255;
        motorPID.SetTunings(Kp, 0, Kd);  // Disable integral action
    } else if (output < 0) {
        output = 0;
        motorPID.SetTunings(Kp, 0, Kd);  // Disable integral action
    } else {
        motorPID.SetTunings(Kp, Ki, Kd);  // Restore integral action
    }
```

b. A back-calculation technique

i. The Back-Calculation Anti-Windup technique dynamically adjusts the integral term to compensate for actuator saturation. Instead of freezing the integral term like clamping, back-calculation feeds the excess output (saturation error) back into the integral term, allowing smoother recovery when the controller output exits saturation. The correction term is calculated using:

$$e^i = e + \frac{(u' - u)}{Tt}$$

where u′ is the unsaturated PID output, u is the actual saturated output, and Tt is the tracking time constant, often set equal to the integral time constant Ti.

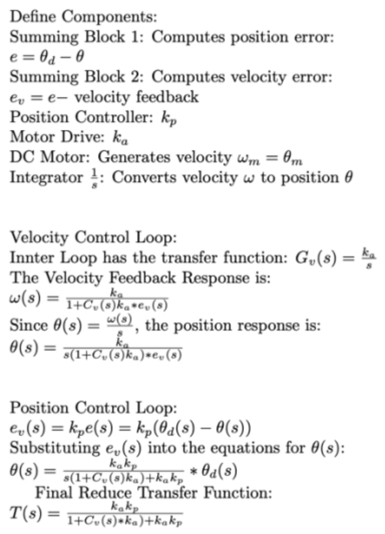
```
double unsat_output = output;
if (output > 255) output = 255;
else if (output < 0) output = 0;

double anti_windup_correction = (unsat_output - output) / Ki;
static double integral_term = 0;
integral_term += anti_windup_correction;
input += integral_term;
```

c. A hybrid clamping/back-calculation technique.

i. A Hybrid Clamping/Back-Calculation technique combines the simplicity of Clamping with the smooth correction of Back-Calculation, ensuring both quick response and stability. In this method, when the output is saturated, the integral term is temporarily clamped, preventing excessive accumulation, but a back-calculation correction factor is also applied to allow for controlled recovery. The hybrid approach prevents integral windup while ensuring smooth transitions when leaving saturation. The implementation would involve freezing the integral term like clamping, but still applying a back-calculation correction

```
    double unsat_output = output;
    if (output > 255) {
        output = 255;
    motorPID.SetTunings(Kp, 0, Kd);  // Disable integral action
} else if (output < 0) {
    output = 0;
    motorPID.SetTunings(Kp, 0, Kd);  // Disable integral action
} else {
    motorPID.SetTunings(Kp, Ki, Kd);  // Restore integral action
}

// Apply back-calculation correction
double anti_windup_correction = (unsat_output - output) / Ki;
static double integral_term = 0;
integral_term += anti_windup_correction;
input += integral_term;
```

12. **Now, let's say we wanted to do position control to make a servo motor instead. Note to make a truly useful servo system we would need a current measurement as well. However, using what we have we can still do position control…. This link https://ebookcentral.proquest.com/lib/ugalib/reader.action?docID=1446445&ppg=530 should take you to page 499 of the book "Mechatronics: A Foundation Course". On this page are two strategies for implementing position control.**
    a. **What is the topology of Figure 7.21?**
        i. The topology is a position control system with nested velocity and position feedback loops. This is a cascade control system, meaning it has an inner velocity control loop and an outer position control loop.
    b. **Use your block diagram reduction skills to reduce both block diagrams to a final closed-loop transfer function.**



Define Components:
Summing Block 1: Computes position error:
$e = \theta_d - \theta$
Summing Block 2: Computes velocity error:
$e_v = e -$ velocity feedback
Position Controller: k_p
Motor Drive: k_a
DC Motor: Generates velocity $\omega_m = \theta_m$
Integrator $\frac{1}{s}$: Converts velocity ω to position θ

Velocity Control Loop:
Innter Loop has the transfer function: $G_v(s) = \frac{k_a}{s}$
The Velocity Feedback Response is:
$\omega(s) = \frac{k_a}{1+C_v(s)k_a * e_v(s)}$
Since $\theta(s) = \frac{\omega(s)}{s}$, the position response is:
$\theta(s) = \frac{k_a}{s(1+C_v(s)k_a) * e_v(s)}$

Position Control Loop:
$e_v(s) = k_p e(s) = k_p(\theta_d(s) - \theta(s))$
Substituting $e_v(s)$ into the equations for $\theta(s)$:
$\theta(s) = \frac{k_a k_p}{s(1+C_v(s)k_a)+k_a k_p} * \theta_d(s)$
Final Reduce Transfer Function:
$T(s) = \frac{k_a k_p}{1+C_v(s) * k_a)+k_a k_p}$

        i.
    c. **Compare the two block diagrams. Give pros/cons for each setup.**
        i. The first system just uses position feedback, meaning it compares the desired position (θd) with the actual position (θm) and adjusts the motor using a proportional controller (Kp). This setup is pretty simple and easy

to implement, but it has some downsides—it tends to be slow, can overshoot the target, and might not handle disturbances well. Since there's no control over velocity, the system can be sluggish and might oscillate a lot when trying to correct its position.

ii. The second system, which uses both position and velocity feedback (cascade control), adds an inner velocity loop. This loop helps stabilize the motor’s speed (ωm) before adjusting position (θm), leading to a faster, smoother response with less overshoot. It also handles disturbances better, making it a good choice for applications that need precise movement. However, this approach is more complex—it requires tuning multiple controllers (Kp and τv), and if the velocity loop isn’t set up right, it can cause stability issues. Plus, it needs extra sensors and processing power, which might not be worth it for simpler tasks.

iii. In short, the position-only control is easier but can be unstable, while cascade control is more reliable but takes more effort to set up. If you need quick, accurate movements, cascade control is the way to go. But if precision and speed aren’t a big deal, the simpler position-only setup might be enough.