PID Controller: Integral Limits and Timing Guide

Understanding Anti-windup Protection and Proper Time Calculation

---

1. Integral Limit (Anti-windup Protection)

What is "Windup"?

Windup happens when the integral term grows **huge** because your ROV can't reach the target. Here's a real scenario:

```cpp
// Your ROV is stuck on the bottom, but you want it at surface
target_depth = 0.0;     // Surface
current_depth = 10.0;   // 10 meters deep, stuck on bottom
depth_error = -10.0;    // Large error that won't go away
```

The Problem Without Limits

```cpp
// Every loop cycle (50 times per second):
controller.integral += error * dt;
controller.integral += (-10.0) * 0.02;  // Gets more negative each cycle

// After 10 seconds:
// integral = -10.0 * 0.02 * 500 cycles = -100.0 (HUGE!)

float I_term = controller.ki * controller.integral;
float I_term = 0.1 * (-100.0) = -10.0;  // Massive integral term!
```

What Happens During Windup

1. **ROV gets stuck** (on bottom, against wall, etc.)
2. **Error persists** for long time
3. **Integral keeps growing** every loop cycle
4. **Integral becomes HUGE** (hundreds or thousands)
5. **When ROV breaks free**, integral is so large it causes:
   - Massive overshoot
   - Dangerous oscillations
   - ROV shoots to surface uncontrollably

Real ROV Example

cpp

```cpp
void demonstrateWindup() {
    // ROV stuck at 5m depth, wants to go to 2m
    float current_depth = 5.0;
    float target_depth = 2.0;
    float error = target_depth - current_depth; // -3.0

    // Without integral limit:
    for (int i = 0; i < 1000; i++) {  // 20 seconds at 50Hz
        controller.integral += error * 0.02;  // Keeps growing!
        // After 1000 loops: integral = -3.0 * 0.02 * 1000 = -60.0
    }

    // When ROV finally breaks free from bottom:
    float I_term = 0.1 * (-60.0) = -6.0;  // Huge upward force!
    // ROV rockets to surface uncontrollably!
}
```

Anti-windup Protection Solution

```cpp
// Limit how large integral can grow
controller.integral += error * dt;

// Clamp integral to reasonable limits
if (controller.integral > controller.integral_limit) {
    controller.integral = controller.integral_limit;
}
if (controller.integral < -controller.integral_limit) {
    controller.integral = -controller.integral_limit;
}

// Now integral can never exceed ±50.0 (for example)
```

Why Different Limits for Different Axes

```cpp
// Depth: High limit because buoyancy causes steady errors
depth_pid.integral_limit = 50.0;    // Needs to fight buoyancy drift

// Yaw: Medium limit for handling water currents
yaw_pid.integral_limit = 30.0;      // Currents cause some steady error

// Pitch: Low limit because ROV is naturally pitch-stable
pitch_pid.integral_limit = 15.0;    // Rarely has persistent errors
```

---

2. Previous Time (For Proper Time Calculation)

Why Time Matters in PID

PID control is **time-dependent**. The math requires knowing **how much time passed** between calculations:

```cpp
// Integral: Accumulate error over TIME
integral += error * dt;  // dt = time step

// Derivative: Rate of change over TIME
derivative = (current_error - prev_error) / dt;  // dt = time step
```

The Problem Without Proper Timing

```cpp
// WRONG: Assumes constant time step
void badPIDCalculation() {
    controller.integral += error * 0.02;  // Assumes exactly 20ms

    float derivative = error - controller.prev_error;  // Missing time!
    // What if this loop took 50ms instead of 20ms?
    // Math is completely wrong!
}
```

Arduino Timing Reality

Your Arduino control loop **NEVER** runs at exact timing:

```cpp
void loop() {
    // This loop might take:
    // - 18ms (if sensors read quickly)
    // - 25ms (if I2C is slow)
    // - 30ms (if serial communication happens)
    // - 100ms (if something blocks)

    calculatePID(depth_pid, error, ???);  // What dt to use?
}
```

Proper Time Calculation

cpp

```cpp
float calculatePID(PID& controller, float error, float dt) {
    // Get current time
    unsigned long current_time = millis();

    // Calculate ACTUAL time that passed
    if (controller.prev_time == 0) {
        // First run, can't calculate dt yet
        controller.prev_time = current_time;
        return 0;
    }

    // Real time difference in seconds
    float real_dt = (current_time - controller.prev_time) / 1000.0;

    // Use REAL time for calculations
    controller.integral += error * real_dt;  // Correct integral

    float derivative = (error - controller.prev_error) / real_dt;  // Correct derivative

    // Store for next calculation
    controller.prev_time = current_time;
    controller.prev_error = error;

    return kp * error + ki * controller.integral + kd * derivative;
}
```

Real-World Timing Example

```cpp
void demonstrateTimingImportance() {
    float error = 5.0;

    // Scenario 1: Loop takes 20ms (normal)
    float dt1 = 0.02;
    integral += error * dt1;  // integral += 5.0 * 0.02 = 0.1

    // Scenario 2: Loop takes 100ms (sensor delay)
    float dt2 = 0.1;
    integral += error * dt2;  // integral += 5.0 * 0.1 = 0.5

    // Same error, but 5x different integral contribution!
    // Without proper timing, PID behaves inconsistently
}
```

Why Store prev_time in Each Controller

```cpp
// Each PID controller tracks its own timing
struct PID {
    unsigned long prev_time;  // Each controller's last update time
};

// You might call controllers at different rates:
void loop() {
    // High-speed depth control (every 20ms)
    if (millis() - last_depth_update > 20) {
        calculatePID(depth_pid, depth_error, dt);
        last_depth_update = millis();
    }

    // Slower yaw control (every 50ms)
    if (millis() - last_yaw_update > 50) {
        calculatePID(yaw_pid, yaw_error, dt);
        last_yaw_update = millis();
    }
}
```

Complete Implementation Example

Robust PID Function with Both Features

cpp

```cpp
float calculatePID(PID& controller, float error) {
    unsigned long current_time = millis();

    // Handle first run
    if (controller.prev_time == 0) {
        controller.prev_time = current_time;
        controller.prev_error = error;
        return 0;
    }

    // Calculate actual time step
    float dt = (current_time - controller.prev_time) / 1000.0;

    // Proportional term
    float P_term = controller.kp * error;

    // Integral term with anti-windup
    controller.integral += error * dt;
    controller.integral = constrain(controller.integral,
                                    -controller.integral_limit,
                                     controller.integral_limit);
    float I_term = controller.ki * controller.integral;

    // Derivative term with proper timing
    float derivative = (error - controller.prev_error) / dt;
    float D_term = controller.kd * derivative;

    // Store for next calculation
    controller.prev_time = current_time;
    controller.prev_error = error;

    return P_term + I_term + D_term;
}
```

Complete PID Structure

cpp

```cpp
struct PID {
    // PID gains
    float kp, ki, kd;

    // State variables
    float prev_error;          // For derivative calculation
    float integral;            // Accumulated error
    float integral_limit;      // Anti-windup protection
    unsigned long prev_time;   // For proper timing
    float output_limit;        // Output saturation

    // Constructor for easy setup
    PID(float p, float i, float d, float int_lim, float out_lim)
        : kp(p), ki(i), kd(d), integral(0), prev_error(0),
          integral_limit(int_lim), output_limit(out_lim), prev_time(0) {}
};
```

ROV Controller Setup Example

cpp

```cpp
// Different controllers with appropriate limits
PID depth_pid{2.0, 0.1, 0.5, 50.0, 100.0};   // High integral limit
PID yaw_pid{1.5, 0.05, 0.3, 30.0, 100.0};    // Medium integral limit
PID surge_pid{1.0, 0.02, 0.2, 20.0, 100.0};  // Low integral limit
PID sway_pid{1.0, 0.02, 0.2, 20.0, 100.0};   // Low integral limit
PID pitch_pid{0.8, 0.01, 0.15, 15.0, 60.0};  // Very low integral limit
PID roll_pid{0.8, 0.01, 0.15, 15.0, 60.0};   // Very low integral limit

void controlROV() {
    // Read sensor values
    float depth_error = target_depth - current_depth;
    float yaw_error = target_yaw - current_yaw;
    // ... other errors

    // Calculate control outputs with proper timing and limits
    float depth_output = calculatePID(depth_pid, depth_error);
    float yaw_output = calculatePID(yaw_pid, yaw_error);
    // ... other outputs

    // Apply to thrusters
    applyThrusterMixing(depth_output, yaw_output, /*...*/);
}
```

Why These Features Are Critical for ROV Safety

Without Anti-windup Protection

- **Dangerous uncontrolled movements** when ROV breaks free from obstacles
- **Massive overshoot** leading to surface breaching or bottom crashes
- **Oscillatory behavior** that can damage equipment
- **Unpredictable responses** in confined spaces

Without Proper Timing

- **Inconsistent behavior** - same error gives different responses
- **Incorrect integral accumulation** leading to steady-state errors
- **Wrong derivative calculations** causing instability
- **Performance degradation** during sensor delays or communication interrupts

Benefits of Proper Implementation

- **Predictable and safe operation** in all conditions
- **Consistent response** regardless of loop timing variations
- **Robust performance** when encountering obstacles or disturbances
- **Professional-grade control system** suitable for real-world deployment

---

Common Tuning Guidelines

Integral Limits by Axis Type

- **Depth Control**: 30-60 (fights buoyancy changes)
- **Horizontal Movement**: 15-25 (moderate disturbances)
- **Attitude Control**: 5-20 (naturally stable, low disturbances)

Timing Considerations

- **Control Loop Frequency**: 20-100 Hz for responsive control
- **Sensor Update Rates**: Match PID timing to sensor capabilities
- **Communication Delays**: Account for telemetry and command latency
- **Safety Timeouts**: Implement maximum time between updates

---

Troubleshooting Guide

Signs of Windup Problems

- ROV "jumps" when breaking free from obstacles
- Excessive overshoot in all axes
- Oscillatory motion that doesn't settle
- Control system becomes more aggressive over time

Signs of Timing Problems

- Inconsistent response to same commands
- Control performance varies with system load
- Integral term behaves erratically
- Derivative kicks during sensor delays

Solutions

1. **Implement integral limits** appropriate for each axis
2. **Use actual timing measurements** instead of assumed values
3. **Test in various scenarios** including stuck conditions
4. **Monitor timing performance** and adjust limits accordingly
5. **Add safety features** like maximum output rates and emergency stops

---

This guide provides the essential understanding needed to implement robust PID control systems for underwater ROV applications. Proper implementation of these features is crucial for safe and effective ROV operation.