# Using the AutoTunePID Library

The `AutoTunePID` library is a powerful tool for adaptive PID control in Arduino projects. It features automatic tuning based on methods like **Ziegler-Nichols**, **Cohen-Coon**, **IMC**, **Tyreus-Luyben**, and **Lambda Tuning (CLD)**, as well as manual tuning options. This guide provides a detailed explanation of how to integrate and use the library effectively.

---

## Table of Contents



---

## Initialization

To initialize the `AutoTunePID` controller, you need to specify the **minimum and maximum output values** for the PID controller. Additionally, you can define the **tuning method**, which defaults to `ZieglerNichols`.

### Example

```
#include "AutoTunePID.h"

// Create an instance of AutoTunePID with a specified output range
AutoTunePID pidController(0, 255, TuningMethod::ZieglerNichols);
```

In this example, the `pidController` is configured to output values between **0 and 255** using the **Ziegler-Nichols** tuning method.

---

## Setting the Setpoint

The **setpoint** represents the target value that the system aims to achieve. Use `setSetpoint()` to define it.

### Example

```
pidController.setSetpoint(100.0); // Set the target value to 100
```

This sets the desired system state to a value of **100**.

---

## Selecting the Tuning Method

The **tuning method** determines how the PID gains are calculated. Use `setTuningMethod()` to choose one of the following options:

- `TuningMethod::ZieglerNichols` : A popular method for process control.
- `TuningMethod::CohenCoon` : Useful for processes with significant time delays.
- `TuningMethod::IMC` : Balances robustness and response speed.
- `TuningMethod::TyreusLuyben` : Minimizes overshoot and improves stability.
- `TuningMethod::LambdaTuning` : Uses the Lambda Tuning (CLD) method for systems with dead time.
- `TuningMethod::Manual` : For direct user-defined gains.

### Example

```
pidController.setTuningMethod(TuningMethod::LambdaTuning);
```

This sets the tuning method to **Lambda Tuning (CLD)** for systems with significant dead time.

---

## Manual Tuning

If you prefer **manual tuning**, set the PID gains directly using `setManualGains()`.

### Example

```
pidController.setManualGains(1.0, 0.5, 0.1); // Set Kp, Ki, and Kd
```

This allows precise control over the **proportional**, **integral**, and **derivative gains**.

---

## Input and Output Filtering

**Input and output filtering** smoothens noisy signals, enhancing the controller's performance. Enable filtering and define a smoothing factor ( `alpha` ) between **0.01 and 1.0**. Smaller values result in more smoothing.

- `enableInputFilter(alpha)` : Smooths the input signal.
- `enableOutputFilter(alpha)` : Smooths the output signal.

### Example

```
pidController.enableInputFilter(0.2); // Apply input smoothing with alpha = 0.2
pidController.enableOutputFilter(0.3); // Apply output smoothing with alpha = 0.3
```

These functions improve stability in systems prone to noise.

---

## Anti-Windup

**Anti-windup** prevents the integral term from accumulating excessively when the output is saturated. Use `enableAntiWindup()` to enable or disable this feature and set a threshold.

### Example

```
pidController.enableAntiWindup(true, 0.8); // Enable anti-windup with 80%
threshold
```

This ensures the integral term is constrained when the output approaches its limits.

---

## Updating the Controller

The `update()` function processes the **current input** and calculates the appropriate **output**. Call it within the control loop.

### Example

```
void loop() {
    float sensorValue = analogRead(A0); // Read sensor input
    pidController.update(sensorValue);

    float output = pidController.getOutput();
    analogWrite(3, output); // Send output to the actuator
}
```

This example continuously updates the PID output based on the sensor reading.

---

## Retrieving PID Gains and Output

Retrieve the computed or manually set PID gains and the current output value:

- `getKp()`, `getKi()`, `getKd()`: Access the proportional, integral, and derivative gains.
- `getOutput()`: Access the controller's current output.
- `getKu()`: Retrieve the ultimate gain (Ku) from auto-tuning.
- `getTu()`: Retrieve the oscillation period (Tu) from auto-tuning.

### Example

```cpp
float kp = pidController.getKp();
float ki = pidController.getKi();
float kd = pidController.getKd();

Serial.print("Kp: "); Serial.println(kp);
Serial.print("Ki: "); Serial.println(ki);
Serial.print("Kd: "); Serial.println(kd);
```

This prints the current PID parameters to the Serial Monitor.

---

## Auto-Tuning Behavior

When **auto-tuning** is enabled, the library uses one of the supported methods (e.g., Ziegler-Nichols, Cohen-Coon, IMC, Tyreus-Luyben, or Lambda Tuning) to compute optimal gains. These gains are applied automatically after tuning completes.

### Notes

- Ensure the system can oscillate safely during the tuning process.
- Adjust tuning duration and output limits to match your system's dynamics.

---

## Operational Modes

The library supports **multiple operational modes** to adapt the PID controller's behavior based on the system's needs. Use `setOperationalMode()` to select one of the following modes:

- **Normal**: Standard PID operation.
- **Reverse**: Reverses the error calculation for cooling systems.
- **Hold**: Stops all calculations to save resources.
- **Preserve**: Minimal calculations to keep the system responsive.
- **Tune**: Performs auto-tuning to determine `Tu` and `Ku`.
- **Auto**: Automatically selects the best operational mode based on system behavior.

### Example

```
pidController.setOperationalMode(OperationalMode::Tune); // Set mode to Tune for auto-tuning
```

This sets the operational mode to **Tune**, enabling auto-tuning.

---

## Oscillation Modes

The library supports **three oscillation modes** for auto-tuning, which determine the amplitude of the oscillations during the tuning process:

- **Normal**: Full oscillation ( `MaxOutput - MinOutput` ).
- **Half**: Half oscillation ( `1/2 MaxOutput - 1/2 MinOutput` ).
- **Mild**: Mild oscillation ( `1/4 MaxOutput - 1/4 MinOutput` ).

You can also set the **number of oscillation steps** for auto-tuning. The default steps for each mode are:

- **Normal**: 10 steps.
- **Half**: 20 steps.
- **Mild**: 40 steps.

### Example

```
pidController.setOscillationMode(OscillationMode::Half); // Set oscillation mode to Half
pidController.setOscillationSteps(15); // Set custom oscillation steps (default is 20 for Half mode)
```

This sets the oscillation mode to **Half** and overrides the default steps to **15**.

---

## Example Sketches

### 1. Ziegler-Nichols Example: Temperature Control

```cpp
#include "AutoTunePID.h"
AutoTunePID tempController(0, 255, TuningMethod::ZieglerNichols);

void setup() {
    tempController.setSetpoint(75.0); // Target temperature
    tempController.enableInputFilter(0.1); // Enable input filtering
    tempController.enableAntiWindup(true, 0.8); // Enable anti-windup
    tempController.setOscillationMode(OscillationMode::Normal); // Set
oscillation mode to Normal
    tempController.setOperationalMode(OperationalMode::Tune); // Set operational
mode to Tune
}

void loop() {
    float temp = readTemperature(); // Read temperature sensor
    tempController.update(temp); // Update PID controller
    analogWrite(HEATER_PIN, tempController.getOutput()); // Control heater
    delay(100);
}
```

## 2. Cohen-Coon Example: Motor Speed Control

```cpp
#include "AutoTunePID.h"
AutoTunePID motorController(0, 255, TuningMethod::CohenCoon);

void setup() {
    motorController.setSetpoint(1500); // Target RPM
    motorController.enableInputFilter(0.2); // Enable input filtering
    motorController.enableAntiWindup(true, 0.7); // Enable anti-windup
    motorController.setOscillationMode(OscillationMode::Half); // Set oscillation
mode to Half
    motorController.setOperationalMode(OperationalMode::Tune); // Set operational
mode to Tune
}

void loop() {
    float rpm = readEncoderSpeed(); // Read motor speed
    motorController.update(rpm); // Update PID controller
    analogWrite(MOTOR_PIN, motorController.getOutput()); // Control motor
    delay(100);
}
```

## 3. IMC Example: Pressure Control

```cpp
#include "AutoTunePID.h"
AutoTunePID pressureController(0, 255, TuningMethod::IMC);

void setup() {
    pressureController.setSetpoint(100.0); // Target pressure
    pressureController.enableInputFilter(0.1); // Enable input filtering
    pressureController.enableAntiWindup(true, 0.8); // Enable anti-windup
    pressureController.setOscillationMode(OscillationMode::Normal); // Set
oscillation mode to Normal
    pressureController.setOperationalMode(OperationalMode::Tune); // Set
operational mode to Tune
}

void loop() {
    float pressure = readPressureSensor(); // Read pressure sensor
    pressureController.update(pressure); // Update PID controller
    analogWrite(PUMP_PIN, pressureController.getOutput()); // Control pump
    delay(100);
}
```

## 4. Tyreus-Luyben Example: Chemical Reactor Temperature Control

```cpp
#include "AutoTunePID.h"
AutoTunePID reactorController(0, 255, TuningMethod::TyreusLuyben);

void setup() {
    reactorController.setSetpoint(80.0); // Target reactor temperature
    reactorController.enableInputFilter(0.1); // Enable input filtering
    reactorController.enableAntiWindup(true, 0.8); // Enable anti-windup
    reactorController.setOscillationMode(OscillationMode::Half); // Set
oscillation mode to Half
    reactorController.setOperationalMode(OperationalMode::Tune); // Set
operational mode to Tune
}

void loop() {
    float temp = readReactorTemperature(); // Read reactor temperature
    reactorController.update(temp); // Update PID controller
    analogWrite(HEATER_PIN, reactorController.getOutput()); // Control heater
    delay(100);
}
```

## 5. Lambda Tuning Example: Flow Control

```cpp
#include "AutoTunePID.h"
AutoTunePID flowController(0, 255, TuningMethod::LambdaTuning);

void setup() {
    flowController.setSetpoint(50.0); // Target flow rate
    flowController.enableInputFilter(0.15); // Enable input filtering
    flowController.enableAntiWindup(true, 0.9); // Enable anti-windup
    flowController.setOscillationMode(OscillationMode::Mild); // Set oscillation
mode to Mild
    flowController.setOperationalMode(OperationalMode::Tune); // Set operational
mode to Tune
}

void loop() {
    float flowRate = readFlowSensor(); // Read flow sensor
    flowController.update(flowRate); // Update PID controller
    analogWrite(VALVE_PIN, flowController.getOutput()); // Control valve
    delay(100);
}
```

---

## Summary of Methods

| Method | Description |
|---|---|
| `setSetpoint(float setpoint)` | Sets the target value for the PID controller. |
| `setTuningMethod(TuningMethod)` | Selects the tuning method. |
| `setManualGains(float kp, ki, kd)` | Sets PID gains manually. |
| `enableInputFilter(float alpha)` | Enables input filtering with a smoothing factor. |
| `enableOutputFilter(float alpha)` | Enables output filtering with a smoothing factor. |
| `enableAntiWindup(bool, float)` | Enables anti-windup with an optional threshold. |
| `update(float currentInput)` | Updates the PID controller with the current input. |
| `getOutput()` | Retrieves the computed PID output. |
| `getKp()`, `getKi()`, `getKd()` | Retrieves the PID gains. |
| `getKu()`, `getTu()` | Retrieves the ultimate gain and oscillation period. |
| `setOperationalMode(OperationalMode)` | Sets the operational mode. |
| `setOscillationMode(OscillationMode)` | Sets the oscillation mode for auto-tuning. |
| `setOscillationSteps(int steps)` | Sets the number of oscillation steps for auto-tuning. |

---