C++ Implementation

Pieter P

This page first presents a simple PID implementation in C++, then it adds output clamping and integral anti-windup. Finally, it lists the real-world implementation used in the actual microcontroller code.

Simple implementation

The following listing gives a very basic implementation of a PID controller in C++. It uses the formulas derived on the previous page.

```cpp
#include <cmath>

/// Very basic, mostly educational PID controller with derivative filter.
class PID {
  public:
    /// @param  kp  Proportional gain   @f$ K_p @f$
    /// @param  ki  Integral gain       @f$ K_i @f$
    /// @param  kd  Derivative gain     @f$ K_d @f$
    /// @param  fc  Cutoff frequency    @f$ f_c @f$ of derivative filter in Hz
    /// @param  Ts  Controller sampling time    @f$ T_s @f$ in seconds
    /// The derivative filter can be disabled by setting `fc` to zero.
    PID(float kp, float ki, float kd, float fc, float Ts)
        : kp(kp), ki(ki), kd(kd), alpha(calcAlphaEMA(fc * Ts)), Ts(Ts) {}

    /// Compute the weight factor α for an exponential moving average filter
    /// with a given normalized cutoff frequency `fn`.
    static float calcAlphaEMA(float fn);

    /// Update the controller with the given position measurement `meas_y` and
    /// return the new control signal.
    float update(float reference, float meas_y) {
        // e[k] = r[k] - y[k], error between setpoint and true position
        float error = reference - meas_y;
        // e_f[k] = α e[k] + (1-α) e_f[k-1], filtered error
        float ef = alpha * error + (1 - alpha) * old_ef;
        // e_d[k] = (e_f[k] - e_f[k-1]) / Tₛ, filtered derivative
        float derivative = (ef - old_ef) / Ts;
        // e_i[k+1] = e_i[k] + Tₛ e[k], integral
        float new_integral = integral + error * Ts;

        // PID formula:
        // u[k] = Kp e[k] + Ki e_i[k] + Kd e_d[k], control signal
        float control_u = kp * error + ki * integral + kd * derivative;

        // store the state for the next iteration
        integral = new_integral;
        old_ef = ef;
        // return the control signal
        return control_u;
    }

  private:
    float kp, ki, kd, alpha, Ts;
    float integral = 0;
    float old_ef = 0;
};

float PID::calcAlphaEMA(float fn) {
    if (fn <= 0)
        return 1;
    // α(fₙ) = cos(2πfₙ) - 1 + √( cos(2πfₙ)² - 4 cos(2πfₙ) + 3 )
    const float c = std::cos(2 * float(M_PI) * fn);
    return c - 1 + std::sqrt(c * c - 4 * c + 3);
}
```

Output clamping and anti-windup

We can easily modify the code from the previous section to clamp the output of the controller, and to stop the integral from winding up if the output is already saturated:

```cpp
/// Very basic, mostly educational PID controller with derivative filter, output
/// clamping and integral anti-windup.
class PID {
  public:
    /* ... */

    /// Update the controller with the given position measurement `meas_y` and
    /// return the new control signal.
    float update(float reference, float meas_y) {
        // e[k] = r[k] - y[k], error between setpoint and true position
        float error = reference - meas_y;
        // e_f[k] = α e[k] + (1-α) e_f[k-1], filtered error
        float ef = alpha * error + (1 - alpha) * old_ef;
        // e_d[k] = (e_f[k] - e_f[k-1]) / Tₛ, filtered derivative
        float derivative = (ef - old_ef) / Ts;
        // e_i[k+1] = e_i[k] + Tₛ e[k], integral
        float new_integral = integral + error * Ts;

        // PID formula:
        // u[k] = Kp e[k] + Ki e_i[k] + Kd e_d[k], control signal
        float control_u = kp * error + ki * integral + kd * derivative;

        // Clamp the output
        if (control_u > max_output)
            control_u = max_output;
        else if (control_u < -max_output)
            control_u = -max_output;
        else // Anti-windup
            integral = new_integral;
        // store the state for the next iteration
        old_ef = ef;
        // return the control signal
        return control_u;
    }

  private:
    float kp, ki, kd, alpha, Ts;
    float max_output = 255;
    float integral = 0;
    float old_ef = 0;
};
```

Real-world implementation

In the actual microcontroller code for the motorized fader driver, we make a few changes to the algorithm introduced above:

- We use integer types for the input, setpoint, error and integral.
- For efficiency, the constants K_i and K_d are premultiplied/divided by the factor T_s.
- The output is turned off completely after a given number of cycles of inactivity (no setpoint changes or human interaction), if the error is small enough.

```
/// Standard PID (proportional, integral, derivative) controller. Derivative
/// component is filtered using an exponential moving average filter.
class PID {
  public:
    PID() = default;
    /// @param  kp
    ///         Proportional gain
    /// @param  ki
    ///         Integral gain
    /// @param  kd
    ///         Derivative gain
    /// @param  Ts
    ///         Sampling time (seconds)
    /// @param  fc
    ///         Cutoff frequency of derivative EMA filter (Hertz),
    ///         zero to disable the filter entirely
    PID(float kp, float ki, float kd, float Ts, float f_c = 0,
        float maxOutput = 255)
        : Ts(Ts), maxOutput(maxOutput) {
        setKp(kp);
        setKi(ki);
        setKd(kd);
        setEMACutoff(f_c);
    }

    /// Update the controller: given the current position, compute the control
    /// action.
    float update(uint16_t input) {
        // The error is the difference between the reference (setpoint) and the
        // actual position (input)
        int16_t error = setpoint - input;
        // The integral or sum of current and previous errors
        int32_t newIntegral = integral + error;
        // Compute the difference between the current and the previous input,
        // but compute a weighted average using a factor α ∈ (0,1]
        float diff = emaAlpha * (prevInput - input);
        // Update the average
        prevInput -= diff;

        // Check if we can turn off the motor
        if (activityCount >= activityThres && activityThres) {
            float filtError = setpoint - prevInput;
            if (filtError >= -errThres && filtError <= errThres) {
                errThres = 2; // hysteresis
                integral = newIntegral;
                return 0;
            } else {
                errThres = 1;
            }
        } else {
            ++activityCount;
            errThres = 1;
        }

        bool backward = false;
        int32_t calcIntegral = backward ? newIntegral : integral;

        // Standard PID rule
        float output = kp * error + ki_Ts * calcIntegral + kd_Ts * diff;

        // Clamp and anti-windup
        if (output > maxOutput)
            output = maxOutput;
        else if (output < -maxOutput)
            output = -maxOutput;
        else
            integral = newIntegral;

        return output;
    }

    void setKp(float kp) { this->kp = kp; }               ///< Proportional gain
    void setKi(float ki) { this->ki_Ts = ki * this->Ts; } ///< Integral gain
    void setKd(float kd) { this->kd_Ts = kd / this->Ts; } ///< Derivative gain

    float getKp() const { return kp; }         ///< Proportional gain
    float getKi() const { return ki_Ts / Ts; } ///< Integral gain
    float getKd() const { return kd_Ts * Ts; } ///< Derivative gain

    /// Set the cutoff frequency (-3 dB point) of the exponential moving average
    /// filter that is applied to the input before taking the difference for
    /// computing the derivative term.
    void setEMACutoff(float f_c) {
        float f_n = f_c * Ts; // normalized sampling frequency
        this->emaAlpha = f_c == 0 ? 1 : calcAlphaEMA(f_n);
    }

    /// Set the reference/target/setpoint of the controller.
    void setSetpoint(uint16_t setpoint) {
        if (this->setpoint != setpoint) this->activityCount = 0;
        this->setpoint = setpoint;
    }
    /// @see @ref setSetpoint(int16_t)
    uint16_t getSetpoint() const { return setpoint; }

    /// Set the maximum control output magnitude. Default is 255, which clamps
    /// the control output in [-255, +255].
    void setMaxOutput(float maxOutput) { this->maxOutput = maxOutput; }
    /// @see @ref setMaxOutput(float)
    float getMaxOutput() const { return maxOutput; }

    /// Reset the activity counter to prevent the motor from turning off.
```

```
    void resetActivityCounter() { this->activityCount = 0; }
    /// Set the number of seconds after which the motor is turned off, zero to
    /// keep it on indefinitely.
    void setActivityTimeout(float s) {
        if (s == 0)
            activityThres = 0;
        else
            activityThres = uint16_t(s / Ts) == 0 ? 1 : s / Ts;
    }

    /// Reset the sum of the previous errors to zero.
    void resetIntegral() { integral = 0; }

  private:
    float Ts = 1;                ///< Sampling time (seconds)
    float maxOutput = 255;       ///< Maximum control output magnitude
    float kp = 1;                ///< Proportional gain
    float ki_Ts = 0;             ///< Integral gain times Ts
    float kd_Ts = 0;             ///< Derivative gain divided by Ts
    float emaAlpha = 1;          ///< Weight factor of derivative EMA filter.
    float prevInput = 0;         ///< (Filtered) previous input for derivative.
    uint16_t activityCount = 0; ///< How many ticks since last setpoint change.
    uint16_t activityThres = 0; ///< Threshold for turning off the output.
    uint8_t errThres = 1;        ///< Threshold with hysteresis.
    int32_t integral = 0;        ///< Sum of previous errors for integral.
    uint16_t setpoint = 0;       ///< Position reference.
};
```