

## 2. 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.

```
1  #include <cmath>
2
3  /// Very basic, mostly educational PID controller with derivative filter.
4  class PID {
5  public:
6      /// @param kp Proportional gain    @f$ K_p @f$
7      /// @param ki Integral gain        @f$ K_i @f$
8      /// @param kd Derivative gain      @f$ K_d @f$
9      /// @param fc Cutoff frequency     @f$ f_c @f$ of derivative filter in Hz
10     /// @param Ts Controller sampling time @f$ T_s @f$ in seconds
11     /// The derivative filter can be disabled by setting `fc` to zero.
12     PID(float kp, float ki, float kd, float fc, float Ts)
13         : kp(kp), ki(ki), kd(kd), alpha(calcAlphaEMA(fc * Ts)), Ts(Ts) {}
14
15     /// Compute the weight factor  $\alpha$  for an exponential moving average filter
16     /// with a given normalized cutoff frequency `fn`.
17     static float calcAlphaEMA(float fn);
18
19     /// Update the controller with the given position measurement `meas_y` and
20     /// return the new control signal.
21     float update(float reference, float meas_y) {
22         // e[k] = r[k] - y[k], error between setpoint and true position
23         float error = reference - meas_y;
24         // e_f[k] =  $\alpha$  e[k] + (1- $\alpha$ ) e_f[k-1], filtered error
25         float ef = alpha * error + (1 - alpha) * old_ef;
26         // e_d[k] = (e_f[k] - e_f[k-1]) / Ts, filtered derivative
27         float derivative = (ef - old_ef) / Ts;
28         // e_i[k+1] = e_i[k] + Ts * e[k], integral
29         float new_integral = integral + error * Ts;
30
31         // PID formula:
32         // u[k] = Kp e[k] + Ki e_i[k] + Kd e_d[k], control signal
33         float control_u = kp * error + ki * integral + kd * derivative;
34
35         // store the state for the next iteration
36         integral = new_integral;
37         old_ef = ef;
38         // return the control signal
39         return control_u;
40     }
41
42 private:
43     float kp, ki, kd, alpha, Ts;
44     float integral = 0;
45     float old_ef = 0;
46 };
47
48 float PID::calcAlphaEMA(float fn) {
49     if (fn <= 0)
50         return 1;
51     //  $\alpha(f_n) = \cos(2\pi f_n) - 1 + \sqrt{\cos(2\pi f_n)^2 - 4 \cos(2\pi f_n) + 3}$ 
52     const float c = std::cos(2 * float(M_PI) * fn);
53     return c - 1 + std::sqrt(c * c - 4 * c + 3);
54 }
```

### 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:

```

1  /// Very basic, mostly educational PID controller with derivative filter, output
2  /// clamping and integral anti-windup.
3  class PID {
4  public:
5      /* ... */
6
7      /// Update the controller with the given position measurement `meas_y` and
8      /// return the new control signal.
9      float update(float reference, float meas_y) {
10         // e[k] = r[k] - y[k], error between setpoint and true position
11         float error = reference - meas_y;
12         // e_f[k] =  $\alpha$  e[k] + (1- $\alpha$ ) e_f[k-1], filtered error
13         float ef = alpha * error + (1 - alpha) * old_ef;
14         // e_d[k] = (e_f[k] - e_f[k-1]) / T_s, filtered derivative
15         float derivative = (ef - old_ef) / Ts;
16         // e_i[k+1] = e_i[k] + T_s e[k], integral
17         float new_integral = integral + error * Ts;
18
19         // PID formula:
20         // u[k] = K_p e[k] + K_i e_i[k] + K_d e_d[k], control signal
21         float control_u = kp * error + ki * integral + kd * derivative;
22
23         // Clamp the output
24         if (control_u > max_output)
25             control_u = max_output;
26         else if (control_u < -max_output)
27             control_u = -max_output;
28         else // Anti-windup
29             integral = new_integral;
30         // store the state for the next iteration
31         old_ef = ef;
32         // return the control signal
33         return control_u;
34     }
35
36 private:
37     float kp, ki, kd, alpha, Ts;
38     float max_output = 255;
39     float integral = 0;
40     float old_ef = 0;
41 };

```

## 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.

```

38  /// Standard PID (proportional, integral, derivative) controller. Derivative
39  /// component is filtered using an exponential moving average filter.
40  class PID {
41  public:
42      PID() = default;
43      /// @param kp
44      /// Proportional gain
45      /// @param ki
46      /// Integral gain
47      /// @param kd
48      /// Derivative gain
49      /// @param Ts
50      /// Sampling time (seconds)
51      /// @param fc
52      /// Cutoff frequency of derivative EMA filter (Hertz),
53      /// zero to disable the filter entirely
54      PID(float kp, float ki, float kd, float Ts, float f_c = 0,
55          float maxOutput = 255)
56          : Ts(Ts), maxOutput(maxOutput) {
57          setKp(kp);
58          setKi(ki);
59          setKd(kd);
60          setEMACutoff(f_c);
61      }
62
63      /// Update the controller: given the current position, compute the control
64      /// action.
65      float update(uint16_t input) {
66          // The error is the difference between the reference (setpoint) and the
67          // actual position (input)
68          int16_t error = setpoint - input;
69          // The integral or sum of current and previous errors
70          int32_t newIntegral = integral + error;
71          // Compute the difference between the current and the previous input,
72          // but compute a weighted average using a factor  $\alpha \in (0,1]$ 
73          float diff = emaAlpha * (prevInput - input);
74          // Update the average
75          prevInput -= diff;
76
77          // Check if we can turn off the motor
78          if (activityCount >= activityThres && activityThres) {
79              float filtError = setpoint - prevInput;
80              if (filtError >= -errThres && filtError <= errThres) {
81                  errThres = 2; // hysteresis
82                  integral = newIntegral;
83                  return 0;
84              } else {
85                  errThres = 1;
86              }
87          } else {
88              ++activityCount;
89              errThres = 1;
90          }
91
92          bool backward = false;
93          int32_t calcIntegral = backward ? newIntegral : integral;
94
95          // Standard PID rule
96          float output = kp * error + ki_Ts * calcIntegral + kd_Ts * diff;
97
98          // Clamp and anti-windup
99          if (output > maxOutput)
100              output = maxOutput;
101          else if (output < -maxOutput)
102              output = -maxOutput;
103          else
104              integral = newIntegral;
105
106          return output;
107      }
108
109      void setKp(float kp) { this->kp = kp; } //< Proportional gain
110      void setKi(float ki) { this->ki_Ts = ki * this->Ts; } //< Integral gain
111      void setKd(float kd) { this->kd_Ts = kd / this->Ts; } //< Derivative gain
112
113      float getKp() const { return kp; } //< Proportional gain
114      float getKi() const { return ki_Ts / Ts; } //< Integral gain
115      float getKd() const { return kd_Ts * Ts; } //< Derivative gain
116
117      /// Set the cutoff frequency (-3 dB point) of the exponential moving average
118      /// filter that is applied to the input before taking the difference for
119      /// computing the derivative term.
120      void setEMACutoff(float f_c) {
121          float f_n = f_c * Ts; // normalized sampling frequency
122          this->emaAlpha = f_c == 0 ? 1 : calcAlphaEMA(f_n);
123      }
124
125      /// Set the reference/target/setpoint of the controller.
126      void setSetpoint(uint16_t setpoint) {
127          if (this->setpoint != setpoint) this->activityCount = 0;
128          this->setpoint = setpoint;
129      }
130      /// @see @ref setSetpoint(int16_t)
131      uint16_t getSetpoint() const { return setpoint; }

```

```

132
133 // Set the maximum control output magnitude. Default is 255, which clamps
134 // the control output in [-255, +255].
135 void setMaxOutput(float maxOutput) { this->maxOutput = maxOutput; }
136 // @see @ref setMaxOutput(float)
137 float getMaxOutput() const { return maxOutput; }
138
139 // Reset the activity counter to prevent the motor from turning off.
140 void resetActivityCounter() { this->activityCount = 0; }
141 // Set the number of seconds after which the motor is turned off, zero to
142 // keep it on indefinitely.
143 void setActivityTimeout(float s) {
144     if (s == 0)
145         activityThres = 0;
146     else
147         activityThres = uint16_t(s / Ts) == 0 ? 1 : s / Ts;
148 }
149
150 // Reset the sum of the previous errors to zero.
151 void resetIntegral() { integral = 0; }
152
153 private:
154     float Ts = 1; //< Sampling time (seconds)
155     float maxOutput = 255; //< Maximum control output magnitude
156     float kp = 1; //< Proportional gain
157     float ki_Ts = 0; //< Integral gain times Ts
158     float kd_Ts = 0; //< Derivative gain divided by Ts
159     float emaAlpha = 1; //< Weight factor of derivative EMA filter.
160     float prevInput = 0; //< (Filtered) previous input for derivative.
161     uint16_t activityCount = 0; //< How many ticks since last setpoint change.
162     uint16_t activityThres = 0; //< Threshold for turning off the output.
163     uint8_t errThres = 1; //< Threshold with hysteresis.
164     int32_t integral = 0; //< Sum of previous errors for integral.
165     uint16_t setpoint = 0; //< Position reference.
166 };

```