Preprints, 4th IFAC Conference on
Advances in Proportional-Integral-Derivative Control
Almería, Spain | June 12-14, 2024

Reference Implementation of the PID Controller [⋆]

E. Sundström [*] T. Hägglund [*] M. Bauer [*] J. Eker [*]
K. Soltesz [*]

[*] *Lund University, Dept. Automatic Control, Lund, Sweden (email: {emil.sundstrom, tore.hagglund, margret.bauer, johan.eker, kristian.soltesz}@control.lth.se)*

Abstract: The PID controller is the by far most frequently employed type of controller. As you read, billions of digitally implemented PID controllers are running, shaping the dynamic behavior of anything from the fan speed in your laptop to safety-critical components in nuclear power plants. Given the abundance of commissioned PID controllers, it is surprisingly hard to find a single source that provides a well-documented, and motivated reference implementation of the PID controller in text-based code. This work provides one. We use the incremental (velocity) form, motivated by its intrinsic integrator anti-windup and bumpless transfer behavior. In addition, we discuss our implementation in terms of measurement filtering, setpoint handling, and runtime environment, among other implementation aspects. Our reference implementation is a living "document", and a link to a GitHub repository hosting the latest version is provided.

Keywords: PID control, Implementation, Programming code.

1 INTRODUCTION

The PID controller is found in a multitude of applications, including most industries: automotive, building climate, medicine, aviation, and consumer electronics. It is wildly recognized and by far the dominating type of controller in real-world applications. The textbook control law

$$u(t) = K\left(e(t) + \frac{1}{T_i}\int_0^t e(\tau)\,d\tau + T_d\frac{de(t)}{dt}\right) \qquad (1)$$

is known to almost all engineers. However, (1) is only the tip of the iceberg, and there are many subtleties to consider when implementing the PID controller. While it can be implemented in analog electronics, mechanics, pneumatics, and even biological circuits, we will limit our scope to digital implementations in programming code, purposed to be interpreted or compiled to run on a computer.

There are numerous extensions to the standard controller, including fractional-order controllers (Chen et al., 2009), PID controllers that additionally consider the second derivative of the error (Huba and Vrančič, 2018), etc. These fall outside our scope. Neither do we consider parameter tuning, a topic thoroughly covered in other sources, e.g., O'Dwyer (2009).

But why do we write a paper about the implementation of the PID controller? Although the PID controller has received a fair share of research attentions, with Google scholar providing well over 1.4 million hits for "PID control" (2023), a vast majority of digital implementations are carried out by corporate entities, and are either proprietary, undocumented or both. The implementation of an ideal PID controller is simple. However, many modifications are necessary to ensure that the controller works in practice. To our knowledge, there is currently no clean, concise and freely available reference implementation in code format that comes together with an exhaustive discussion of its features, and underlying considerations. There are excellent university lecture slides and books, e.g., Åström and Hägglund (2006) and Visioli (2006), covering some aspects. The International Society of Automation (ISA) has published a standard (ISA, 2023), but its focus lies on nomenclature and use, rather than implementation.

These circumstances make it difficult for control engineers to find a concise and thoroughly explained reference implementation. As a result, many PID implementations in commercial solutions do not include important features required to deal with practical problems.

Here we present a reference implementation. One key contribution is the discussions explaining why it looks the way it does, which we motivate through discussing it in the contexts of

- Generality of parameterization
- Enabling of common use-cases
- Measurement filtering
- Setpoint weighting
- Integrator anti-windup
- Bumpless transfer
- Runtime system considerations

The implementation provided here thus deals with the most commonly found practical problems and goes beyond the simple discretisation of the PID algorithm.

[⋆] This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. All authors are members of the ELLIIT Strategic Research Area.

2 REFERENCE IMPLEMENTATION

The proposed PID controller implementation is schematically depicted in figure 1. All variables (parameters, signals, states) used in the code are listed alongside brief explanations and references to the sections, in table 1 of section 5 at the end of this article. Functions are listed in table 2 of section 5.

Before listing the code of our reference implementation in section 2.3, some comments about the pseudo code programming language are given in section 2.1, and the assumed runtime environment is given in section 2.2. The code is then further explained and discussed in section 3.

2.1 Programming language

Our pseudo code format is intended to be accessible to anyone with some programming background, while also being easily portable to commonly used programming languages. The few special constructs that we use are explained below.

Functions (definitions and calls) are signaled with parenthesis enclosing a comma-separated list of one or several arguments. Any arguments after a possible semicolon in the argument list are optional. They default to the zero of the corresponding type (e.g., 0.0, false, none), unless another default is explicitly given on the right-hand-side of an equal sign.

The keyword `persistent` signals that local variables, following in a comma-separated list, survive between consecutive calls of the function scope in which they are defined. In the persistent declaration, equal signs after variable names indicate initial values, assigned during the first execution of the statement (but not in consecutive executions). If an initial value is not explicitly declared, the variable will be initialized to its zero type (cf. optional argument).

A single layer of local scoping is assumed for functions, but not for branching constructs such as `if` blocks.

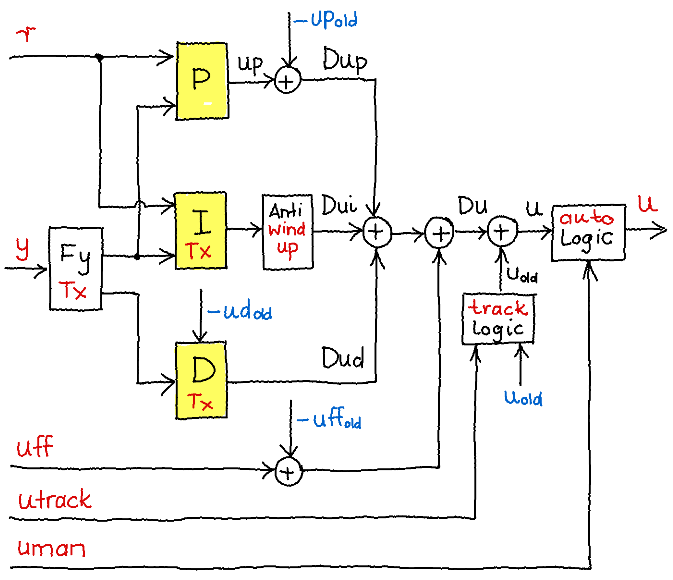

Fig. 1. Schematic of our PID controller implementation.

Code comments are indicated by a leading hash mark `#`. Keywords of our pseudo code format are typeset in blue.

2.2 Runtime environment

All functions are assumed to be non-reentrant, meaning that once a function has been invoked, consecutive calls to it (possibly from another runtime thread) are blocked until the function has returned.

An abstraction consisting of user-defined set and get functions are used to communicate with external system components, such as hardware. For example `get_y()` will retrieve the latest available measurement signal, while `set_u(u)` will issue the control signal `u`.

While the discussed functionalities of our language and runtime model are common to many languages, the actual implementation might need slight modifications from the reference that we are about to present, depending e.g., on the variable scoping model or how parameters can be protected from other threads in critical sections.

2.3 Code listings

In this section, the code of our reference implementation is listed, and very briefly commented. More in-depth analysis of the code and discussions about choices, such as the use of incremental form, are provided in section 3. All code is made available at `https://github.com/copybit/pid` (commit 9ac19a8).

The main component of our implementation is the function `PID` of listing 1. It is invoked with setpoint (reference) `r` and measurement[1] `y` values as its main arguments. The optional Boolean argument `auto` sets the controller in automatic (auto) mode when true, and in manual (man) mode when false. When `auto==false`, the control signal is given by the input `uman`. The optional flag `windup` encodes its possible binary values `none` (00), `upper` (01), `lower` (10), and `both` (11). This flag enables externally triggering integrator anti-windup, as motivated in section 3.7. If the Boolean argument `track` is true, the controller output is forced to follow the external signal `utrack`[2]. This is performed by adding the output increment `Du` to the current value of `utrack` instead of to the previously calculated value `u_old`. The optional argument `uff` is the feed-forward control signal from an external feed-forward controller. Finally, the optional argument `Tx`, with default value 1.0, constitutes a scaling factor for controller and filter gains, to compensate for variable execution cycle duration.

The integrator anti-windup function `anti_windup()`, called from line 32 of listing 1, is provided in listing 4. Filter implementation are provided in listing 2. Zero-order-hold (ZOH) discretization of the measurement filter is given in listing 3. Finally, listing 5 illustrates how a run-method for the PID controller could look in a threaded run-time environment.

[1] Note that the measurement input to the controller does not necessarily have to be a (sensor) measurement—it could for example be a control signal of another controller in a mid-ranging control structure, and the output of the controller does not necessarily have to be a control signal—it could for example be the setpoint to another controller. Alternative names are provided in e.g., ISA (2023).

[2] Tracking mode can only be enabled when `auto==true`.

```
PID(r,y;uff,uman,utrack,Tx=1.0,
        track,auto,windup)

  #Initialize parameter states
  persistent kp,ki,kd,umin,umax,u0,b=1

  #Initialize signal states
  persistent u_old,up_old,ud_old,uff_old

  #Filter updates
  yf,dyf=Fy(y,Tx)

  if auto
    if ki==0
      u_old=u0 #Bias term if P or PD control
      up_old=0
      ud_old=0
      uff_old=0
      b=1
    end

    if track #Tracking mode
      u_old=utrack
      up_old=0
      ud_old=0
      uff_old=0
    end

    #Control signal increments
    Dup=kp*(b*r-yf)-up_old
    Dui=ki*(r-yf)*Tx
    Dui=anti_windup(Dui,windup)
    Dud=(-kd*dyf-ud_old)/Tx
    Duff=uff-uff_old

    #Add control signal increment
    Du=Dup+Dui+Dud+Duff
    u=u_old+Du
  else
    u=uman #Manual control signal
  end

  #Saturate and send control signal
  u=max(min(u,umax),umin)
  set_u(u)

  #Update old signal states
  u_old=u
  up_old=kp*(b*r-yf)
  ud_old=-kd*dyf
  uff_old=uff
end
```

Listing 1. PID update, discussed in section 3.

```
yf,dyf=Fy(y;Tx=1.0)
  persistent TfTs=10.0 #Tf per nominal Ts
  persistent a11,a12,a21,a22,b1,b2 #Params
  persistent yf,dyf,Tx_old #Filter state

  #Rediscretize to match execution period
  if not(Tx==Tx_old)
    a11,a12,a21,a22,b1,b2=zoh_Fy(TfTs,Tx)
  end

  #State update
  yf1=yf
  Tx_old=Tx
  yf=a11*yf1+a12*dyf+b1*y
  dyf=a21*yf1+a22*dyf+b2*y
end
```

Listing 2. Measurement filtering, discussed in section 3.4.

```
a11,a12,a21,a22,b1,b2=zoh_Fy(TfTs;Tx=1.0)
  #Help variables
  h1=Tx/TfTs
  h2=exp(-h1)
  h3=h1*h2
  h4=h3/TfTs

  #Filter parameters
  a11=h2+h3
  a12=h2
  a21=-h4
  a22=h2-h3
  b1=1-h2-h3
  b2=h4
end
```

Listing 3. Filter discretization, discussed in section 3.4.

```
Dui=anti_windup(Dui,windup)
  #Prevent increase, decrease, or both
  if windup==both or windup==lower
    Dui=max(Dui,0)
  end
  if windup==both or windup==upper
    Dui=min(Dui,0)
  end
end
```

Listing 4. External anti-windup, discussed in section 3.7.

```
run(Ts=1.0)
  #Set true to stop execution
  persistent stop

  #Initialize internal state
  persistent auto,track,windup,t_old=time(),
             Ts=Ts

  while not(stop)
    #Time when loop starts
    t0=time()

    #Read signals from runtime or hardware
    r=get_r()
    y=get_y()
    uff=get_uff()
    utrack=get_utrack()
    uman=get_uman()

    #Compute time between two executions
    t=time()
    Tx=(t-t_old)/Ts

    #Invoke the PID update
    PID(r,y,uff,uman,utrack,Tx,
            track,auto,windup)

    t_old=t #State update
    sleep(Ts-(t-t0)) #(Non-blocking) sleep
  end
end
```

Listing 5. Example of run method for periodic execution of the PID controller of listing 1 in a runtime thread.

3 DISCUSSION

The remainder of this paper discusses aspects of the reference implementation provided in section 2.3.

3.1 *Incremental control law*

The controller must be discretized to be implementable in a digital system. Equation (1) is straight-forward to ZOH-discretize, since this corresponds to a Riemann (rectangle) approximation of the integral. The discretized gains are given as `kp`, `ki`, `kd` in listing 1.

After the PID control law in (1) has been discretized, it is in the so-called direct form. In each iteration, the control signal is computed from the most recent reference and measurement signals, with the integral of the error as the controller state.

In contrast, the incremental form computes the control signal increment since the last iteration (`Du` in listing 1), and uses the control signal itself to encode the controller state. The direct form is sometimes referred to as positional form, and the incremental as velocity form. This is natural if we regard the control signal as a position, in which case the increments will be (scaled) ZOH approximations of the associated speed.

The control law for the incremental form is

$$u(t) = u(t - T_s) + \Delta u_p + \Delta u_i + \Delta u_d + \Delta u_{ff}, \qquad (2)$$

where Δu_p represents the increment in applied control signal from the proportional part (row 30 in listing 1), Δu_i the increment from the integral part (row 32 in listing 1), and Δu_d the increment from the derivative part (row 33 in listing 1). $u(t - T_s)$ is the control signal from the last iteration and Δu_{ff} represents the increment in external feed-forward control signal. T_s is the time since the last iteration, discussed in section 3.5.

The parametrization of the controller can be done in several ways. For example, the textbook PID controller parametrization in K, T_i, and T_d of (1) is known as parallel parametrization (Åström and Hägglund, 2006), and the linear parametrization form is

$$u(t) = k_p e(t) + k_i \int_0^t e(\tau)\, d\tau + k_d \frac{de(t)}{dt}, \qquad (3)$$

in k_p, k_i, k_d. Confusingly, this is sometimes also referred to as parallel. We use (3) since it is more general than the series and parallel forms. The transition between these parametrizations is done by:

$$\begin{bmatrix} k_p & k_i & k_d \end{bmatrix} = \begin{bmatrix} K & \dfrac{K}{T_i} & KT_d \end{bmatrix}. \qquad (4)$$

Computing the increments between iterations relies on storing previous proportional part `up_old`, derivative part `ud_old` and feed-forward part `uff_old` of the control signal. On the other hand, the incremental form has two important advantages over the direct form, that we will discuss in sections 3.6 and 3.7.

3.2 *Setpoint-free derivative*

With the control laws considered so far, a step in setpoint r will result in a corresponding step in error $e = r - y$. With derivative action, $k_d > 0$ in (3), there will thus be an impulse in the control signal u. Favoring a smooth control signal, we will employ the common modification of excluding the reference from the derivative part, replacing $\dot{e}(t)$ with $-\dot{y}(t)$. This is of course not mandatory, since there are some applications where the reference signal preferably shall be used in the derivative part.

3.3 *Setpoint weighting*

Another measure to obtain a smoother control signal is to introduce a setpoint weight b, and replace the error $e(t)$ in the proportional part with $br(t) - y(t)$. If $b = 1$, the nominal behavior of (3) is attained. With $b = 0$, a change in setpoint is only introduced into the control signal u of (3) through the integrator, that still needs to be computed based on the actual error $e = r - y$ (and not $br - y$). Note that when no integral term is present, that is when the controller is a P or PD controller, the setpoint weight must be $b = 1$.

3.4 *Measurement filtering*

It is not only reference changes that can generate undesirably high control signal activity, but also measurement noise. To mitigate this, it is advisable to low-pass filter the measurement signal. Note that the filtering is implemented as external functions (listing 2 and listing 3), so the second-order low-pass filter proposed in this paper can easily be replaced by another filter if desired.

While a first-order low-pass filter makes derivative action implementable, a second-order filter provides the controller with high frequency roll off, that is, the controller gain asymptotically goes to zero for high frequencies. Although a first-order filter would suffice for controllers without derivative action, there is no disadvantage of using a second-order filter. Second-order filters have slightly larger phase loss, but not to an extent that makes a practical difference in the context of PID control.

We use a critically damped second-order filter to avoid resonant modes

$$Y_f(s) = \frac{1}{(sT_f+1)^2}Y(s), \tag{5}$$

that relates the measurement $Y(s)$ to its filtered counterpart $Y_f(s)$ in the Laplace domain. The time constant T_f is a parameter that needs to be determined alongside k_p, k_i, k_d of (3).

Selecting the filtered measurement $y_f(t)$ and its time derivative $\dot{y}_f(t)$ as state variables, (5) can be expressed as the (continuous) time-domain state-space form:

$$\begin{bmatrix}\dot{y}_f\\ \ddot{y}_f\end{bmatrix} = \underbrace{\begin{bmatrix}0 & 1\\ -\frac{1}{T_f^2} & -\frac{2}{T_f}\end{bmatrix}}_{A_c}\begin{bmatrix}y_f\\ \dot{y}_f\end{bmatrix} + \underbrace{\begin{bmatrix}0\\ \frac{1}{T_f^2}\end{bmatrix}}_{B_c} y. \tag{6}$$

The discretization of the filter is performed with zero-order-hold (ZOH). For a sampling period T_s and the continuous system 6, the discretized filter is

$$A = \begin{bmatrix}h_2+h_3 & T_s h_2\\ -h_4 & h_2-h_3\end{bmatrix}, \tag{7a}$$

$$B = \begin{bmatrix}1-h_2-h_3\\ h_4\end{bmatrix}, \tag{7b}$$

where the help variables

$$\begin{bmatrix}h_1 & h_2 & h_3 & h_4\end{bmatrix} = \begin{bmatrix}\frac{T_s}{T_f} & \exp(-h_1) & h_1 h_2 & \frac{h_3}{T_f}\end{bmatrix} \tag{8}$$

can be sequentially computed from T_f, T_s. This, and subsequent computations of the filter coefficient (matrix entries) of (7), are performed in listing 3. Update of the measurement filter outputs is performed in listing 2.

3.5 Jitter compensation

If the controller can not run periodically, we have implemented a compensation for possible 'jitter'. Internally, time is re-scaled in each iteration so that $T_s = 1$ in scaled time unit, and T_x is the scale factor to achieve this. This is why there is no T_s explicitly in listing 3. The first argument `TfTs` is not the filter time constant T_f expressed in the internal time unit as `TfTs=Tf/Ts` when the controller is setup, not wall-time. Wall-time is the time from start to end of a program.

3.6 Bumpless transfer

In most applications, the control signal is not allowed to jump when changing from manual to automatic mode, from tracking to non-tracking mode, or when changing parameters of the controller. In direct form implementations, this can be achieved by keeping track of when such changes occur, and changing the integrator state to a value that results in a bumpless transfer. The incremental form controller state is encoded in the control signal, eliminating the need for such checks.

3.7 Integrator anti-windup

There are two common ways to avoid integrator windup in PID controllers: tracking and clamping. In practice they both work satisfactory, but tracking requires tuning an additional parameter. We have therefore chosen clamping for our reference implementation. In the increnental form, it is extremely simple to implement clamping by just saturating the control signal, as done on line 44 of listing 1.

listing 4 implements the possibility to prevent windup manually by setting the input `windup` to a desired value. There are scenarios when the integration shall only be allowed in one direction, which is why listing 4 allows this behaviour. In other scenarios, there is no need for external anti-windup functionality, and listing 4 can be removed without further consequences to the rest of the implementation.

3.8 Additional run-time considerations

There are other aspects beside algorithmic issues. For example, we have not explained how to achieve thread-protection of shared variables. This varies between runtime enviroments, as does system-specific aspects of implementing the various `set_` and `get_` functions. Further issues concern numeric aspects, such as signal scaling, that are particularly important if fixed point arithmetic is dictated by hardware or timing requirements. Furthermore, (small) performance gains can be made by removing functionality that will not be used, such as e.g., the capability to handle variable update intervals through `Tx`.

4 Conclusions

In this article, the pseudo-code of a practical PID implementation has been presented. The implementation considers the most common practical issues such as integral windup, bumpless transfer, filtering and jitter compensation. Both feedforward and tracking of external signals are included in the code. We have used the incremental (velocity) form, motivated by its intrinsic integrator anti-windup and bumpless behavior at mode switches and controller parameter updates.

As mentioned before, this reference implementation is a living "document", and the link to the GitHub repository hosting the latest version is
`https://github.com/copybit/pid` (commit 9ac19a8).

5 Nomenclature

Table 1. Variables of listings 1 to 5 in alphabetic order; reference to section they are discussed in; brief explanations.

Variable	Section	Explanation
a11,a12,a21,a22	3.4	Discrete-time measurement filter system matrix elements.
auto	3.6	Boolean flag set to true in auto mode; false in manual mode.
b	3.3	Setpoint weight.
b1,b2	3.4	Discrete-time measurement filter input matrix elements.
Du	3.1	Control signal increment.
Duff	3.1	Feed-forward control signal increment.
Dup,Dui,Dud	3.1	Control signal term increments.
dyf	3.4	Filtered measurement derivative.
h1,h2,h3,h4	3.4	Help variables to encode measurement filter discretization.
kp,ki,kd	3.1	Linear form controller gains.
r	3.3	Setpoint (reference).
stop	-	Boolean flag to stop execution.
t0,t,t_old	3.5	Wall time stamps at beginning of run function of listing 5; just before invoking the PID update; just before invoking previous PID update.
TfTs	3.5	Number of filter time constants per nominal sampling period: Tf/Ts.
track	-	Boolean flag indicating tracking mode.
Ts	3.5	Nominal PID sampling time.
Tx	3.5	Scale factor that relates nominal update period Tx to actual: Tx*Ts.
Tx_old	3.5	Previous value of Tx.
u	3.1	Control signal.
u0	3.3	Bias term in P and PD control.
uff	3.1	Feedforward control signal.
uff_old	3.1	Previous feed-forward control signal.
uman	3.6	Manual control signal.
umin,umax	3.7	Control signal saturation limits.
up_old,ud_old	3.1	P- and D-values from last iteration.
utrack	-	Tracking signal.
u_old	3.1	Previous control signal. State in the controller.
windup	3.7	Externally generated windup flag.
y	3.4	Measurement signal.
yf	3.4	Filtered measurement signal.
yf1	3.4	Help variable in the filter update function.

Table 2. Functions used in our reference implementation (listings 1 to 5) in alphabetic order; reference to section they are discussed in; brief explanations.

Function	Section	Explanation
PID		listing 1: PID controller core code.
anti_windup	3.7	listing 4: External integrator anti-windup.
zoh_Fy	3.4	listing 3: Zero-order-hold (ZOH) discretization of measurement filter.
exp	-	Returns the exponent of its argument.
Fy	3.4	listing 2: Measurement filter.
get_r	3.8	Reads the setpoint.
get_uff	3.8	Reads feed-forward control signal.
get_uman	3.8	Reads manual control signal.
get_utrack	3.8	Reads tracking signal.
get_y	3.8	Reads measurement signal.
min, max	3.7	Take two arguments of an ordered type, and return the min (max) vale.
not	-	Logical not, negates it boolean argument.
set_u	3.8	Sets the control signal.
sleep	-	(Non-blocking) sleep, where the argument is time in units of the system wall time.
time	3.5	Returns system wall time stamp.

References

Åström, K. and Hägglund, T. (2006). *Advanced PID Control.* ISA - The Instrumentation, Systems and Automation Society, Research Triangle Park, North Carolina.

Chen, Y., Petras, I., and Xue, D. (2009). Fractional order control - a tutorial. In *2009 American Control Conference*, 1397–1411. doi:10.1109/ACC.2009.5160719.

Huba, M. and Vrančič, D. (2018). Comparing filtered PI, PID and PIDD2 control for the FOTD plants. *IFAC-PapersOnLine*, 51(4), 954–959. 3rd IFAC Conference on Advances in Proportional-Integral-Derivative Control PID 2018.

ISA (2023). ISA-TR5.9-2023, Proportional-Integral-Derivative (PID) Algorithms and Performance. Technical report, International Society of Automation.

O'Dwyer, A. (2009). *Handbook of PI and PID Controller Tuning Rules.* Imperial College Press, London, UK.

Visioli, A. (2006). *Practical PID Control.* Springer, Berlin.