
micropython-simple-pid

Release 1.1.0

Martin Lundberg, Jorge Marques'

Jan 17, 2022

MODULES: INTRO SIMPLE_{PID}.RST

1	MicroPython simple-pid	1
1.1	Installation	1
1.2	Usage	1
1.3	License	3
2	Indices and tables	5

MICROPYTHON SIMPLE-PID

A simple and easy to use PID controller in MicroPython. If you want a PID controller without external dependencies that just works, this is for you! The PID was designed to be robust with help from [Brett Beauregards guide](#) and ported to MicroPython from the [Python simple-pid](#) version.

Usage is very simple:

```
from PID import PID
pid = PID(1, 0.1, 0.05, setpoint=1, scale='us')

# Assume we have a system we want to control in controlled_system
v = controlled_system.update(0)

while True:
    # Compute new output from the PID according to the systems current value
    control = pid(v)

    # Feed the PID output to the system and get its current value
    v = controlled_system.update(control)
```

Complete API documentation can be found [here](#).

1.1 Installation

Upload the `simple_pid/PID.py` file to your board running MicroPython, either to the root path or the `lib` path (as `lib/PID.py`).

1.2 Usage

The PID class implements `__call__()`, which means that to compute a new output value, you simply call the object like this:

```
output = pid(current_value)
```

1.2.1 The basics

The PID works best when it is updated at regular intervals. To achieve this, set `sample_time` to the amount of time there should be between each update and then call the PID every time in the program loop. A new output will only be calculated when `sample_time` in the setted scale has passed:

```
pid.sample_time = 10 # Update every 10 milliseconds when set to the millisecond (ms)
↪ scale

while True:
    output = pid(current_value)
```

To set the scale of the controller, add the following when initiating the controller, accepted values are 's' for seconds, 'ms' for milliseconds, 'us' for microseconds, 'ns' for nanoseconds and 'cpu' for the highest precision at cpu clock. On default and on error set to seconds.

```
pid = PID(1, 0.1, 0.05, setpoint=1, scale='cpu')
```

To set the setpoint, ie. the value that the PID is trying to achieve, simply set it like this:

```
pid.setpoint = 10
```

The tunings can be changed any time when the PID is running. They can either be set individually or all at once:

```
pid.Ki = 1.0
pid.tunings = (1.0, 0.2, 0.4)
```

To use the PID in *reverse mode*, meaning that an increase in the input leads to a decrease in the output (like when cooling for example), you can set the tunings to negative values:

```
pid.tunings = (-1.0, -0.1, 0)
```

Note that all the tunings should have the same sign.

In order to get output values in a certain range, and also to avoid *integral windup* (since the integral term will never be allowed to grow outside of these limits), the output can be limited to a range:

```
pid.output_limits = (0, 10) # Output value will be between 0 and 10
pid.output_limits = (0, None) # Output will always be above 0, but with no upper bound
```

1.2.2 Other features

Auto mode

To disable the PID so that no new values are computed, set auto mode to False:

```
pid.auto_mode = False # No new values will be computed when pid is called
pid.auto_mode = True # pid is enabled again
```

When disabling the PID and controlling a system manually, it might be useful to tell the PID controller where to start from when giving back control to it. This can be done by enabling auto mode like this:

```
pid.set_auto_mode(True, last_output=8.0)
```

This will set the I-term to the value given to `last_output`, meaning that if the system that is being controlled was stable at that output value the PID will keep the system stable if started from that point, without any big bumps in the output when turning the PID back on.

Observing separate components

When tuning the PID, it can be useful to see how each of the components contribute to the output. They can be seen like this:

```
p, i, d = pid.components # The separate terms are now in p, i, d
```

Proportional on measurement

To eliminate overshoot in certain types of systems, you can calculate the [proportional term directly on the measurement](#) instead of the error. This can be enabled like this:

```
pid.proportional_on_measurement = True
```

Error mapping

To transform the error value to another domain before doing any computations on it, you can supply an `error_map` callback function to the PID. The callback function should take one argument which is the error from the setpoint. This can be used e.g. to get a degree value error in a yaw angle control with values between $[-\pi, \pi]$:

```
import math

def pi_clip(angle):
    if angle > 0:
        if angle > math.pi:
            return angle - 2*math.pi
    else:
        if angle < -math.pi:
            return angle + 2*math.pi
    return angle

pid.error_map = pi_clip
```

1.3 License

Licensed under the [MIT License](#).

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`