

 **mike-matera / FastPID** Public

A fast, integer based PID controller suitable for Arduino.

 LGPL-2.1 license

 111 stars  36 forks

 Star

 Notifications

 Code

 Issues 2

 Pull requests


 Actions

 Projects

 Wiki

 Security



 master

Go to file

Mike Matera Update ignore to skip CLion files. ...

on Mar 26, 2019  87

[View code](#)

 README.md

FastPID

A fast 32-bit fixed-point PID controller for Arduino

About

This PID controller is faster than alternatives for Arduino because it avoids expensive floating point operations. The PID controller is configured with floating point coefficients and translates them to fixed point internally. This imposes limitations on the domain of the coefficients. Setting the I and D terms to zero makes the controller run faster. The controller is configured to run at a fixed frequency and calling code is responsible for running at that frequency. The k_i and k_d parameters are scaled by the frequency to save time during the `step()` operation.

Description of Coefficients

- k_p - P term of the PID controller.
- k_i - I term of the PID controller.

- K_d - D term of the PID controller.
- H_z - The execution frequency of the controller.

Coefficient Domain

The computation pipeline expects 16 bit coefficients. This is controlled by `PARAM_BITS` and should not be changed or calculations may overflow. The number of bits before and after the decimal place is controlled by `PARAM_SHIFT` in `FastPID.h`. The default value for `PARAM_SHIFT` is 8 and can be changed to suit your application.

- The parameter P domain is [0.00390625 to 255] inclusive.
- The parameter I domain is P / H_z
- The parameter D domain is $P * H_z$

The controller checks for parameter domain violations and won't operate if a coefficient is outside of the range. All of the configuration operations return `bool` to alert the user of an error. The `err()` function checks the error condition. Errors can be cleared with the `clear()` function.

Execution Frequency

The execution frequency is not automatically detected as of version v1.1.0 This greatly improves the controller performance. Instead the K_i and K_d terms are scaled in the configuration step. It's essential to call `step()` at the rate that you specify.

Input and Output

The input and the setpoint are an `int16_t` this matches the width of Analog pins and accomodate negative readings and setpoints. The output of the PID is an `int16_t`. The actual bit-width and signedness of the output can be configured.

- `bits` - The output width will be limited to values inside of this bit range. Valid values are 1 through 16
- `sign` If `true` the output range is $[-2^{(bits-1)}, -2^{(bits-1)}-1]$. If `false` output range is $[0, 2^{(bits-1)}-1]$. **The maximum output value of the controller is 32767 (even in 16 bit unsigned mode)**

Performance

FastPID performance varies depending on the coefficients. When a coefficient is zero less calculation is done. The controller was benchmarked using an Arduino UNO and the code below.

Kp	Ki	Kd	Step Time (uS)
0.1	0.5	0.1	~64
0.1	0.5	0	~56
0.1	0	0	~28

For comparison the excellent [ArduinoPID](#) library takes an average of about 90-100 uS per step with all non-zero coefficients.

API

The API strives to be simple and clear. I won't implment functions in the controller that would be better implemented outside of the controller.

`FastPID()`

Construct a default controller. The default coefficients are all zero. Do not use a default-constructed controller until after you've called `setCoefficients()` and `setOutputconfig()`

```
FastPID(float kp, float ki, float kd, float hz, int bits=16, bool sign=false)
```

Construct a controller that's ready to go. Calls the following:

```
configure(kp, ki, kd, hz, bits, sign);
```

```
bool setCoefficients(float kp, float ki, float kd, float hz);
```

Set the PID coefficients. The coefficients `ki` and `kd` are scaled by the value of `hz`. The `hz` value informs the PID of the rate you will call `step()`. Calling code is responsible for calling `step()` at the rate in `hz`. Returns `false` if a configuration error has occurred. Which could be from a previous call.

```
bool setOutputConfig(int bits, bool sign);
```

Set the output configuration by bits/sign. The output range will be:

For signed equal to `true`

- $2^{(n-1)} - 1$ down to $-2^{(n-1)}$

For signed equal to `false`

- $2^n - 1$ down to 0

Bits equals 16 is a special case. When bits is 16 and sign is `false` the output range is

- 32767 down to 0

Returns `false` if a configuration error has occurred. Which could be from a previous call.

```
bool setOutputRange(int16_t min, int16_t max);
```

Set the output range directly. The effective range is:

- Min: -32768 to 32766
- Max: -32767 to 32767

Min must be greater than max.

Returns `false` if a configuration error has occurred. Which could be from a previous call.

```
void clear();
```

Reset the controller. This should be done before changing the configuration in any way.

```
bool configure(float kp, float ki, float kd, float hz, int bits=16, bool sign=false)
```



Bulk configure the controller. Equivalent to:

```
clear();
setCoefficients(kp, ki, kd, hz);
setOutputConfig(bits, sign);
```

```
int16_t step(int16_t sp, int16_t fb);
```

Run a single step of the controller and return the next output.

```
bool err();
```

Test for a configuration error. The controller will not run if this function returns `true`.

Integral Windup

Applications that control slow moving systems and have a non-zero integral term often see significant overshoot on startup. This is caused by the integral sum "winidng up" as it remembers a long time away from the setpoint. If this describes your system there are two things you can do.

Addressing Windup: Limit the Sum

There are constants in `FastPID.h` that control the maximum allowable integral. Lowering these prevents the controller from remembering as much offset from the setpoint and will reduce the overshoot.

```
#define INTEG_MAX    (INT32_MAX)
#define INTEG_MIN    (INT32_MIN)
```

Change these constants with caution. Setting them too low will fix your overshoot problem but it will negatively affect the controller's ability to regulate the load. If you're unsure of the right constant use the next solution instead of limiting the sum.

Limiting Windup: Bounded Regulation

The PID controller works best when the system is close to the setpoint. During the startup phase, or in the case of a significant excursion, you can disable PID control entirely. An example of this can be found in the Sous-Vide controller example in this project. The core of the logic is in this code:

```
if (feedback < (setpoint * 0.9)) {
    analogWrite(PIN_OUTPUT, 1);
    myPID.clear();
}
else {
    analogWrite(PIN_OUTPUT, myPID.step(setpoint, feedback));
}
```

The code bypasses the PID when the temperature is less than 90% of the setpoint, simply turning the heater on. When the temperature is above 90% of the setpoint the PID is enabled. Fixing your overshoot this way gives you much better control of your system without having to add complex, invalid and difficult to understand features to the PID controller.

Sample Code

```
#include <FastPID.h>

#define PIN_INPUT      A0
#define PIN_SETPOINT   A1
#define PIN_OUTPUT     9

float Kp=0.1, Ki=0.5, Kd=0.1, Hz=10;
int output_bits = 8;
bool output_signed = false;

FastPID myPID(Kp, Ki, Kd, Hz, output_bits, output_signed);

void setup()
{
    Serial.begin(9600);
    if (myPID.err()) {
        Serial.println("There is a configuration error!");
        for (;;) {}
    }
}

void loop()
{
    int setpoint = analogRead(PIN_SETPOINT) / 2;
    int feedback = analogRead(PIN_INPUT);
    int ts = micros();
    uint8_t output = myPID.step(setpoint, feedback);
    int tss = micros();
    analogWrite(PIN_OUTPUT, output);
    Serial.print("(Fast) micros: ");
    Serial.print(tss - ts);
    Serial.print(" sp: ");
    Serial.print(setpoint);
    Serial.print(" fb: ");
    Serial.print(feedback);
    Serial.print(" out: ");
    Serial.println(output);
}
```

```
    delay(100);  
}
```

Releases 8

[+ 7 releases](#)

Packages

No packages published

Contributors 3



mike-matera Mike Matera



adammck Adam Mckaig



per1234

Languages

● Python 50.2% ● C++ 44.9% ● Makefile 4.9%