Project Blog

Project updates and... um...

« Introducing Proportional On Measurement

Proportional on Measurement – The Code

In the previous post I spent all my time explaining the benefits of Proportional on Measurement. In this post I'll explain the code. People seemed to appreciate the step-by-step way I explained things last time, so that's what I'll do here. The 3 passes below detail how I went about adding PonM to the PID library.

First Pass – Initial Input and Proportional-Mode selection

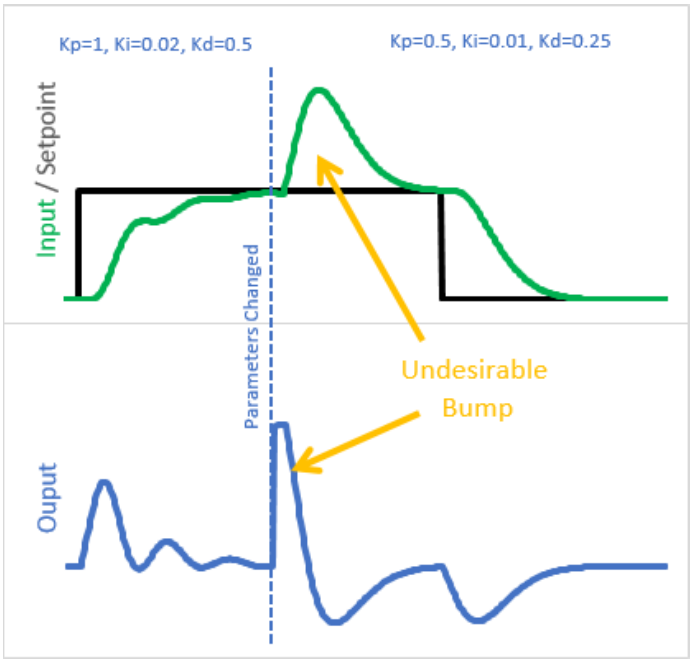
```
1  /*working variables*/
2  unsigned long lastTime;
3  double Input, Output, Setpoint;
4  double ITerm, lastInput;
5  double kp, ki, kd;
6  int SampleTime = 1000; //1 sec
7  double outMin, outMax;
8  bool inAuto = false;
9
10 #define MANUAL 0
11 #define AUTOMATIC 1
12
13 #define DIRECT 0
14 #define REVERSE 1
15 int controllerDirection = DIRECT;
16
17 #define P_ON_M 0
18 #define P_ON_E 1
19 bool pOnE = true;
20 double initInput;
21
22 void Compute()
```

Proportional on Measurement provides increasing resistance as the input changes, but without a frame of reference our performance would be a little wonky. If the PID Input is 10000 when we first turn on the controller, do we really want to start resisting with Kp*10000? No. We want to use our initial input as a reference point (line 108,) the increase or decrease resistance as the input changes from there (line 38.)

The other thing we need to do is allow the user to select whether they want to do Proportional on Error or Measurement. After the last post it might seem like PonE is useless, but it's important to remember that for many loops it works well. As such, we need to let the user choose which mode they want (lines 51&55) and then act accordingly in the calculation (lines 37&38).

Second Pass – On-the-fly tuning changes

While the code above does indeed work, it has a problem that we've seen before. When tuning parameters are changed at runtime, we get an undesirable blip.



Why is this happening?

Output Spikes

	Output	=	Kp	*	(Input - initInput)	+	Iterm	-	Kd	*	dInput
Just Before	7.01		-1		29.37		36.38		-0.5		-0.003
Just After	21.7		-0.5		29.36		36.38		-0.25		-0.004

Because resistance from P-Term is suddenly halved

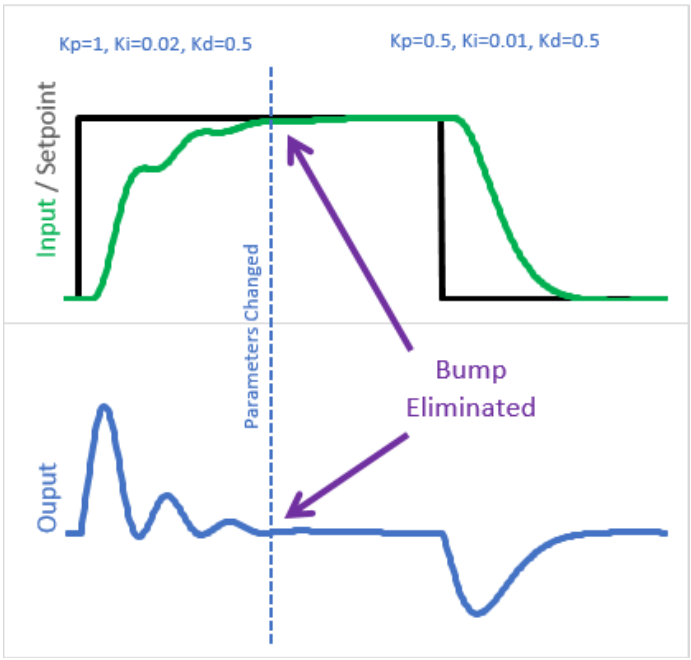
The last time we saw this, it was that the integral was being rescaled by a new Ki. This time, it's that (Input – initInput) is being rescaled by Kp. the solution I chose is similar to what I did for Ki: instead of treating the Input – initInput as a monolithic unit multiplied by the current Kp, I broke it into individual steps multiplied by the Kp at that time:

$$-K_P[\text{Input}_n - \text{Input}_{init}] = -K_{P_n}[\text{Input}_n - \text{Input}_{n-1}] - \ \ldots -K_p[\text{Input}_1 - \text{Input}_{init}]$$

```
1  /*working variables*/
2  unsigned long lastTime;
3  double Input, Output, Setpoint;
4  double ITerm, lastInput;
5  double kp, ki, kd;
6  int SampleTime = 1000; //1 sec
7  double outMin, outMax;
8  bool inAuto = false;
9
10 #define MANUAL 0
11 #define AUTOMATIC 1
12
13 #define DIRECT 0
14 #define REVERSE 1
15 int controllerDirection = DIRECT;
16
17 #define P_ON_M 0
18 #define P_ON_E 1
19 bool pOnE = true;
20 double PTerm;
21
22 void Compute()
```

Instead of multiplying the entirety of Input-initInput by Kp, we now keep a working sum, PTerm. At each step we multiply just the current input change by Kp and subtract it from PTerm (line 41.)

Here we can see the impact of the change:



No Output Bump

	Output	= PTerm	+ Iterm	- Kd	* dInput
Just Before	7.01	-29.37	36.38	-0.5	-0.003
Just After	7.02	-29.36	36.38	-0.25	-0.004

Because Now Kp only affects us moving forward

Because the old Kps are “in the bank”, the change in tuning parameters only affects us moving forward

Final Pass – Sum problems.

I won’t go into complete detail (fancy trends etc) as to what wrong with the above code. It’s pretty good, but there are still major issues with it. For example:

1. **Windup, sort of**: While the final output is limited between outMin and outMax, it’s possible for PTerm to grow when it shouldn’t. It wouldn’t be as bad as integral windup, but it still wouldn’t be acceptable
2. **On-the-fly changes**: If the user were to change from P_ON_M to P_ON_E while running, then after some time return back, PTerm wouldn’t be property initialized and it would cause an output bump

There are more, but just these are enough to see what the real issue is. We’ve dealt with all of these before, back when we created ITerm. Rather than go through and re-implement the same solutions for PTerm, I decided on a more aesthetically-pleasing solution.

By merging PTerm and ITerm into a single variable called “outputSum”, the P_ON_M code then benefits from the all the ITerm fixes that are already in place, and because there aren’t two sums in the code there isn’t needless redundancy.

```
1  /*working variables*/
2  unsigned long lastTime;
3  double Input, Output, Setpoint;
4  double outputSum, lastInput;
5  double kp, ki, kd;
6  int SampleTime = 1000; //1 sec
```

```
 7  double outMin, outMax;
 8  bool inAuto = false;
 9
10  #define MANUAL 0
11  #define AUTOMATIC 1
12
13  #define DIRECT 0
14  #define REVERSE 1
15  int controllerDirection = DIRECT;
16
17  #define P_ON_M 0
18  #define P_ON_E 1
19  bool pOnE = true;
20
21
```

And there you have it. The above functionality is what is now present in v1.2.0 of the Arduino PID.

But wait, there's more: Setpoint Weighting.

I didn't add the following to the Arduino library code, but this is a feature that might be of some interest if you want to roll your own. Setpoint Weighting is, at it's core, a way to have both PonE and PonM at the same time. By specifying a ratio between 0 and 1, you can have 100% PonM, 100% PonE (respectively) or some ratio in between. This can be helpful if you have a process that's not perfectly integrating (like a reflow oven) and want to account for this.

Ultimately I decided not to add it to the library at this time, as it winds up being ANOTHER parameter to adjust/explain, and I didn't think the resulting benefit was worth it. At any rate, here is the code if you'd like to modify the code to have Setpoint Weighting instead of just pure PonM/PonE selection:

```
 1  /*working variables*/
 2  unsigned long lastTime;
 3  double Input, Output, Setpoint;
 4  double outputSum, lastInput;
 5  double kp, ki, kd;
 6  int SampleTime = 1000; //1 sec
 7  double outMin, outMax;
 8  bool inAuto = false;
 9
10  #define MANUAL 0
11  #define AUTOMATIC 1
12
13  #define DIRECT 0
14  #define REVERSE 1
15  int controllerDirection = DIRECT;
16
17  #define P_ON_M 0
18  #define P_ON_E 1
19  bool pOnE = true, pOnM = false;
20  double pOnEKp, pOnMKp;
21
22
```

Instead of setting pOn as an integer, it now comes in as a double which allows for a ratio (line 58.) In addition to some flags (lines 62&63) weighted Kp terms are computed on lines 77-78. Then on lines 37 and 43 the weighted PonM and PonE contributions are added to the overall PID output.

This entry was posted on Tuesday, June 20th, 2017 at 10:32 am and is filed under Coding, PID. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response, or trackback from your own site.

Leave a Reply

Name (required)

Mail (will not be published) (required)

Website

Submit Comment

- Search for: Search

- **Links**
  - 
  - 
  - 

- **This Site**
  - About
  - Project Index

- **Categories**
  - PID (25)
    - Coding (12)
    - Front End (2)
    - Showcase (4)

- Projects (40)
  - Craft (6)
  - Electronic (12)
  - Mechanical (26)
- Uncategorized (4)

Archives

- June 2017
- November 2012
- September 2012
- July 2012
- June 2012
- April 2012
- March 2012
- January 2012
- December 2011
- October 2011
- September 2011
- August 2011
- July 2011
- June 2011
- May 2011
- April 2011
- September 2010
- August 2010
- July 2010
- March 2010
- November 2009
- October 2009
- September 2009
- August 2009
- July 2009
- June 2009
- May 2009

Project Blog is proudly powered by WordPress
Entries (RSS) and Comments (RSS).