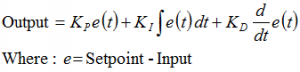
**The Beginner’s PID**

Here’s the PID equation as everyone first learns it:

[](http://brettbeauregard.com/blog/wp-content/uploads/2011/03/pidalgorithm.png)

This leads pretty much everyone to write the following PID controller:

|  |  |
| --- | --- |
| 3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | /\*working variables\*/  unsigned long lastTime;  double Input, Output, Setpoint;  double errSum, lastErr;  double kp, ki, kd;  void Compute()  {     /\*How long since we last calculated\*/     unsigned long now = millis();     double timeChange = (double)(now - lastTime);       /\*Compute all the working error variables\*/     double error = Setpoint - Input;     errSum += (error \* timeChange);     double dErr = (error - lastErr) / timeChange;       /\*Compute PID Output\*/     Output = kp \* error + ki \* errSum + kd \* dErr;       /\*Remember some variables for next time\*/     lastErr = error;     lastTime = now;  }    void SetTunings(double Kp, double Ki, double Kd)  {     kp = Kp;     ki = Ki;     kd = Kd;  } |

Compute() is called either regularly or irregularly, and it works pretty well. This series isn’t about “works pretty well” though. If we’re going to turn this code into something on par with industrial PID controllers, we’ll have to address a few things:

1. [**Sample Time –**](http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/improving-the-beginner%E2%80%99s-pid-sample-time) The PID algorithm functions best if it is evaluated at a regular interval. If the algorithm is aware of this interval, we can also simplify some of the internal math.
2. [**Derivative Kick –**](http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/improving-the-beginner%E2%80%99s-pid-derivative-kick) Not the biggest deal, but easy to get rid of, so we’re going to do just that.
3. [**On-The-Fly Tuning Changes –**](http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/improving-the-beginner%E2%80%99s-pid-tuning-changes) A good PID algorithm is one where tuning parameters can be changed without jolting the internal workings.
4. [**Reset Windup Mitigation –**](http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/improving-the-beginner%E2%80%99s-pid-reset-windup)We’ll go into what Reset Windup is, and implement a solution with side benefits
5. [**On/Off (Auto/Manual) –**](http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/improving-the-beginner%E2%80%99s-pid-onoff) In most applications, there is a desire to sometimes turn off the PID controller and adjust the output by hand, without the controller interfering
6. [**Initialization –**](http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/improving-the-beginner%E2%80%99s-pid-initialization) When the controller first turns on, we want a “bumpless transfer.” That is, we don’t want the output to suddenly jerk to some new value
7. [**Controller Direction –**](http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/improving-the-beginners-pid-direction) This last one isn’t a change in the name of robustness per se. it’s designed to ensure that the user enters tuning parameters with the correct sign.
8. [**NEW: Proportional on Measurement –**](http://brettbeauregard.com/blog/2017/06/proportional-on-measurement-the-code/) Adding this feature makes it easier to control certain types of processes

**The Problem**

The Beginner’s PID is designed to be called irregularly. This causes 2 issues:

* You don’t get consistent behavior from the PID, since sometimes it’s called frequently and sometimes it’s not.
* You need to do extra math computing the derivative and integral, since they’re both dependent on the change in time.

**The Solution**

Ensure that the PID is called at a regular interval. The way I’ve decided to do this is to specify that the compute function get called every cycle. based on a pre-determined Sample Time, the PID decides if it should compute or return immediately.

Once we know that the PID is being evaluated at a constant interval, the derivative and integral calculations can also be simplified. Bonus!

**The Code**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | /\*working variables\*/  unsigned long lastTime;  double Input, Output, Setpoint;  double errSum, lastErr;  double kp, ki, kd;  int SampleTime = 1000; //1 sec  void Compute()  {     unsigned long now = millis();     int timeChange = (now - lastTime);     if(timeChange>=SampleTime)     {        /\*Compute all the working error variables\*/        double error = Setpoint - Input;        errSum += error;        double dErr = (error - lastErr);          /\*Compute PID Output\*/        Output = kp \* error + ki \* errSum + kd \* dErr;          /\*Remember some variables for next time\*/        lastErr = error;        lastTime = now;     }  }    void SetTunings(double Kp, double Ki, double Kd)  {    double SampleTimeInSec = ((double)SampleTime)/1000;     kp = Kp;     ki = Ki \* SampleTimeInSec;     kd = Kd / SampleTimeInSec;  }    void SetSampleTime(int NewSampleTime)  {     if (NewSampleTime > 0)     {        double ratio  = (double)NewSampleTime                        / (double)SampleTime;        ki \*= ratio;        kd /= ratio;        SampleTime = (unsigned long)NewSampleTime;     }  } |

On lines 10&11, the algorithm now decides for itself if it’s time to calculate. Also, because we now KNOW that it’s going to be the same time between samples, we don’t need to constantly multiply by time change. We can merely adjust the Ki and Kd appropriately (lines 31 & 32) and result is mathematically equivalent, but more efficient.

one little wrinkle with doing it this way though though. if the user decides to change the sample time during operation, the Ki and Kd will need to be re-tweaked to reflect this new change. that’s what lines 39-42 are all about.

Also Note that I convert the sample time to Seconds on line 29. Strictly speaking this isn’t necessary, but allows the user to enter Ki and Kd in units of 1/sec and s, rather than 1/mS and mS.

**The Results**

the changes above do 3 things for us

1. Regardless of how frequently Compute() is called, the PID algorithm will be evaluated at a regular interval [Line 11]
2. Because of the time subtraction [Line 10] there will be no issues when millis() wraps back to 0. That only happens every 55 days, but we’re going for bulletproof remember?
3. We don’t need to multiply and divide by the timechange anymore. Since it’s a constant we’re able to move it from the compute code [lines 15+16] and lump it in with the tuning constants [lines 31+32]. Mathematically it works out the same, but it saves a multiplication and a division every time the PID is evaluated

**Side note about interrupts**

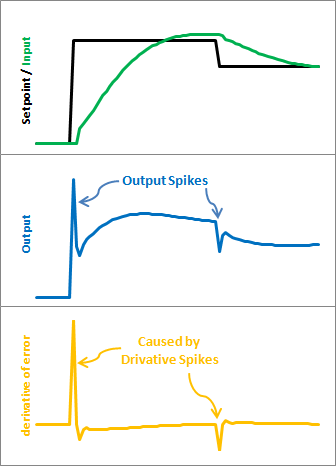
If this PID is going into a microcontroller, a very good argument can be made for using an interrupt. SetSampleTime sets the interrupt frequency, then Compute gets called when it’s time. There would be no need, in that case, for lines 9-12, 23, and 24. If you plan on doing this with your PID implentation, go for it! Keep reading this series though. You’ll hopefully still get some benefit from the modifications that follow.  
There are three reasons I didn’t use interrupts

1. As far as this series is concerned, not everyone will be able to use interrupts.
2. Things would get tricky if you wanted it implement many PID controllers at the same time.
3. If I’m honest, it didn’t occur to me. [Jimmie Rodgers](http://jimmieprodgers.com/) suggested it while proof-reading the series for me. I may decide to use interrupts in future versions of the PID library.

III/

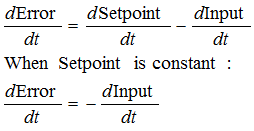
**The Problem**

This modification is going to tweak the derivative term a bit. The goal is to eliminate a phenomenon known as “Derivative Kick”.

[](http://brettbeauregard.com/blog/wp-content/uploads/2011/03/DonE.png)

The image above illustrates the problem. Since error=Setpoint-Input, any change in Setpoint causes an instantaneous change in error. The derivative of this change is infinity (in practice, since dt isn’t 0 it just winds up being a really big number.) This number gets fed into the pid equation, which results in an undesirable spike in the output. Luckily there is an easy way to get rid of this.

**The Solution**

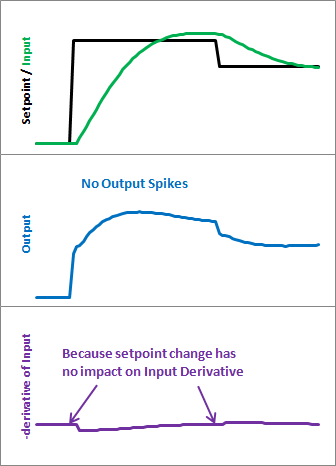
[](http://brettbeauregard.com/blog/wp-content/uploads/2011/03/DonMExplain.png)  
It turns out that the derivative of the Error is equal to negative derivative of Input, EXCEPT when the Setpoint is changing. This winds up being a perfect solution. Instead of adding (Kd \* derivative of Error), we subtract (Kd \* derivative of Input). This is known as using “Derivative on Measurement”

**The Code**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | /\*working variables\*/  unsigned long lastTime;  double Input, Output, Setpoint;  double errSum, lastInput;  double kp, ki, kd;  int SampleTime = 1000; //1 sec  void Compute()  {     unsigned long now = millis();     int timeChange = (now - lastTime);     if(timeChange>=SampleTime)     {        /\*Compute all the working error variables\*/        double error = Setpoint - Input;        errSum += error;        double dInput = (Input - lastInput);          /\*Compute PID Output\*/        Output = kp \* error + ki \* errSum - kd \* dInput;          /\*Remember some variables for next time\*/        lastInput = Input;        lastTime = now;     }  }    void SetTunings(double Kp, double Ki, double Kd)  {    double SampleTimeInSec = ((double)SampleTime)/1000;     kp = Kp;     ki = Ki \* SampleTimeInSec;     kd = Kd / SampleTimeInSec;  }    void SetSampleTime(int NewSampleTime)  {     if (NewSampleTime > 0)     {        double ratio  = (double)NewSampleTime                        / (double)SampleTime;        ki \*= ratio;        kd /= ratio;        SampleTime = (unsigned long)NewSampleTime;     }  } |

The modifications here are pretty easy. We’re replacing +dError with -dInput. Instead of remembering the lastError, we now remember the lastInput

**The Result**

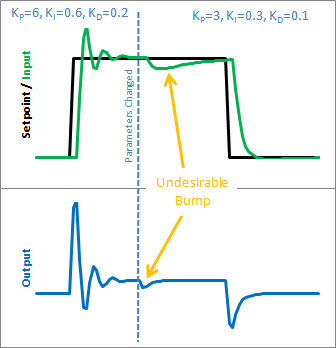
[](http://brettbeauregard.com/blog/wp-content/uploads/2011/03/DonM.png)

Here’s what those modifications get us. Notice that the input still looks about the same. So we get the same performance, but we don’t send out a huge Output spike every time the Setpoint changes.

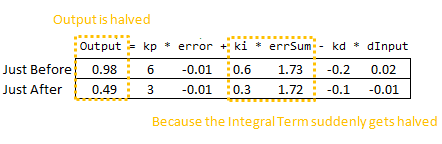
IV/

**The Problem**

The ability to change tuning parameters while the system is running is a must for any respectable PID algorithm.

[](http://brettbeauregard.com/blog/wp-content/uploads/2011/03/BadIntegral.png)

The Beginner’s PID acts a little crazy if you try to change the tunings while it’s running. Let’s see why. Here is the state of the beginner’s PID before and after the parameter change above:

[](http://brettbeauregard.com/blog/wp-content/uploads/2011/03/BadIntegralCode.png)

So we can immediately blame this bump on the Integral Term (or “I Term”). It’s the only thing that changes drastically when the parameters change. Why did this happen? It has to do with the beginner’s interpretation of the Integral:

[http://brettbeauregard.com/blog/wp-content/uploads/2011/03/BadIntegralEqn1.png](http://brettbeauregard.com/blog/wp-content/uploads/2011/03/BadIntegralEqn1.png)

This interpretation works fine until the Ki is changed. Then, all of a sudden, you multiply this new Ki times the entire error sum that you have accumulated. That’s not what we wanted! We only wanted to affect things moving forward!

**The Solution**

There are a couple ways I know of to deal with this problem. The method I used in the last library was to rescale errSum. Ki doubled? Cut errSum in Half. That keeps the I Term from bumping, and it works. It’s kind of clunky though, and I’ve come up with something more elegant. (There’s no way I’m the first to have thought of this, but I did think of it on my own. That counts damnit!)

The solution requires a little basic algebra (or is it calculus?)

[http://brettbeauregard.com/blog/wp-content/uploads/2011/03/GoodIntegralEqn.png](http://brettbeauregard.com/blog/wp-content/uploads/2011/03/GoodIntegralEqn.png)

Instead of having the Ki live outside the integral, we bring it inside. It looks like we haven’t done anything, but we’ll see that in practice this makes a big difference.

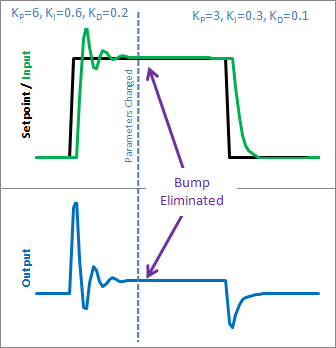
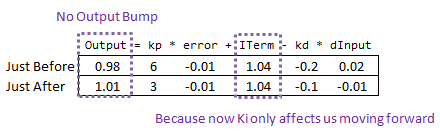
Now, we take the error and multiply it by whatever the Ki is at that time. We then store the sum of THAT. When the Ki changes, there’s no bump because all the old Ki’s are already “in the bank” so to speak. We get a smooth transfer with no additional math operations. It may make me a geek but I think that’s pretty sexy.

**The Code**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | /\*working variables\*/  unsigned long lastTime;  double Input, Output, Setpoint;  double ITerm, lastInput;  double kp, ki, kd;  int SampleTime = 1000; //1 sec  void Compute()  {     unsigned long now = millis();     int timeChange = (now - lastTime);     if(timeChange>=SampleTime)     {        /\*Compute all the working error variables\*/        double error = Setpoint - Input;        ITerm += (ki \* error);        double dInput = (Input - lastInput);          /\*Compute PID Output\*/        Output = kp \* error + ITerm - kd \* dInput;          /\*Remember some variables for next time\*/        lastInput = Input;        lastTime = now;     }  }    void SetTunings(double Kp, double Ki, double Kd)  {    double SampleTimeInSec = ((double)SampleTime)/1000;     kp = Kp;     ki = Ki \* SampleTimeInSec;     kd = Kd / SampleTimeInSec;  }    void SetSampleTime(int NewSampleTime)  {     if (NewSampleTime > 0)     {        double ratio  = (double)NewSampleTime                        / (double)SampleTime;        ki \*= ratio;        kd /= ratio;        SampleTime = (unsigned long)NewSampleTime;     }  } |

So we replaced the errSum variable with a composite ITerm variable [Line 4]. It sums Ki\*error, rather than just error [Line 15]. Also, because Ki is now buried in ITerm, it’s removed from the main PID calculation [Line 19].

**The Result**

[](http://brettbeauregard.com/blog/wp-content/uploads/2011/03/GoodIntegral.png)  
[](http://brettbeauregard.com/blog/wp-content/uploads/2011/03/GoodIntegralCode.png)  
So how does this fix things. Before when ki was changed, it rescaled the entire sum of the error; every error value we had seen. With this code, the previous error remains untouched, and the new ki only affects things moving forward, which is exactly what we want.

V/