Improving the Beginner’s PID - Introduction

翻译：rei1984

In conjunction with the release of the new Arduino PID Library I′ve decided to release this series of posts. The last library, while solid, didn′t really come with any code explanation. This time around the plan is to explain in great detail why the code is the way it is. I′m hoping this will be of use to two groups of people:

- **People directly interested in what′s going on inside the Arduino PID library will get a detailed explanation.**
- **Anyone writing their own PID algorithm can take a look at how I did things and borrow whatever they like.**

It′s going to be a tough slog, but I think I found a not-too-painful way to explain my code. I′m going to start with what I call ʺThe Beginner′s PID.ʺ I′ll then improve it step-by-step until we′re left with an efficient, robust pid algorithm.

为了结合新版 arduino PID 库的发布，我决定发表一个系列的文章。在最新的库里，并没有程序的注解和解释。这次我计划围绕程序做一个详细的解释，为什么程序是这样写的。我希望对以下 2 类人有用：
对 arduino pid 库感兴趣的人，能得到源码的解释。
任何正在写自己 pid 程序的人，能看看我的作品，并且随性浏览。

The Beginner’s PID

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

$$\text{Output} = K_P e(t) + K_I \int e(t)\,dt + K_D \frac{d}{dt} e(t)$$

$$\text{Where : } e = \text{Setpoint - Input}$$

This leads pretty much everyone to write the following PID controller:
下面有 pid 公式，每个人初次学习都会遇到：

$$\text{Output} = K_P e(t) + K_I \int e(t)\,dt + K_D \frac{d}{dt} e(t)$$

$$\text{Where : } e = \text{Setpoint - Input}$$

这就引导了每个人写出如下的 pid 算法：

```
/*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:

Compute（）会被经常或者不经常的调用，他也能运行的很好。这个系列不是讨论工作的怎么好的思路。如果我们用这份代码对比工业上的 **pid** 控制器，我们不得不涉及到如下几件事：

1. **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.
采样时间。如果 **pid** 能有规律的被调用。**Pid** 算法函数能有很好的表现。如果算法在采样间隔内，我们也能简化一些内部计算。

2. **Derivative Kick - Not the biggest deal, but easy to get rid of, so we're going to do just that.**
微分界跃**-**不是最大的难点，很简单的可以去除，所以我们打算那样做。

3. **On-The-Fly Tuning Changes - A good PID algorithm is one where tuning parameters can be changed without jolting the internal workings.**
运行的时候更改 **PID** 参数，不会引起界跃输出。

4. **Reset Windup Mitigation -We'll go into what Reset Windup is, and implement a solution with side benefits**
对 **windup mitigation** 的复位**-**我们介绍什么是复位 **windup**。采用一种方式解决误差。

5. **On/Off (Auto/Manual) - In most applications, there is a desire to sometimes turn off the PID controller and adjust the output by hand, without the controller interfering**
pid 的开关。在很多应用里面，**pid** 内有意的关闭，并且用人工的方式调整输出，没有和控制器连接。

6. **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**
初始化，但我们打开控制器。我们会进行一些参数的传递。这样我们就不会突然输出一个新值。

7. **Controller 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.**
pid 输出的方向 **-** 这是最后一篇，主要是引进符号的概念。增加 **output** 可能 **input** 变小，所以需要符号来表别方向。

Once we've addressed all these issues, we'll have a solid PID algorithm. We'll also, not coincidentally, have the code that's being used in the lastest version of the Arduino PID Library. So whether you're trying to write your own algorithm, or trying to understand what's going on inside the PID library, I hope this helps you out. Let's get started.
Next >>

一旦我们解决了所有问题，我们就有一个坚实的 **pid** 算法。并非巧合，我们会采用最新的 **arduino pid** 的算法。所以，不管是你打算写自己的算法还是把 **pid** 库内容弄明白。我都希望帮助你，让我们开始吧。

UPDATE: In all the code examples I'm using doubles. On the Arduino, a double is the same as a float (single precision.) True double precision is WAY overkill for PID. If the language you're using does true double precision, I'd recommend changing all doubles to floats.

更新：所有的程序实例都采用 double 双精度。在 arduino 中，双精度 double 是采用 float 单精度处理的。Double 对于 pid 来杀鸡用牛刀了。如果你使用的语言有双精度的支持，我建议改变所有的双精度 double 变为单精度 float。

Sample Time

采样时间

(This is Modification #1 in a larger series on writing a solid PID algorithm)

The Problem

The Beginner's PID is designed to be called irregularly. This causes 2 issues:
不规则的调用 **pid** 算法，会产生 **2** 个问题：

- **You don't get consistent behavior from the PID, since sometimes it's called frequently and sometimes it's not.**
  当你有时候规则的调用 **pid** 算法，有时候又停止调用时，你不会得要 **pid** 的一个长期的调整。（算法不能正确的执行）
- **You need to do extra math computing the derivative and integral, since they're both dependent on the change in time.**
  你必须要做微分和积分，因为他们都是和时间息息相关的。

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

```
/*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)//等间隔采样
   {
      /*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;//简化 Ki 计算
   kd = Kd / SampleTimeInSec; //简化 Kd 计
算
}

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 30&31) 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.

恕我怕麻烦，简单的翻译一下，文章比较简单。

1. 通过等间隔时间的判断进行等间隔采样。
2. **Ki Kd** 的参数包含了 ***** SampleTimeInSec (/ SampleTimeInSec),这样不用每次都做乘除计算了

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（bug 已改）]. Mathematically it works out the same, but it saves a multiplication and a division every time the PID is evaluated**

好处，我在上班都已经阐述了。

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 suggested it while proof-reading the series for me. I may decide to use interrupts in future versions of the PID library.**

等间隔采样可以用中断的方式。但是一般还是在 main（） loop（）循环里判断。作者会在将来试试用中断方式测试 pad 算法。

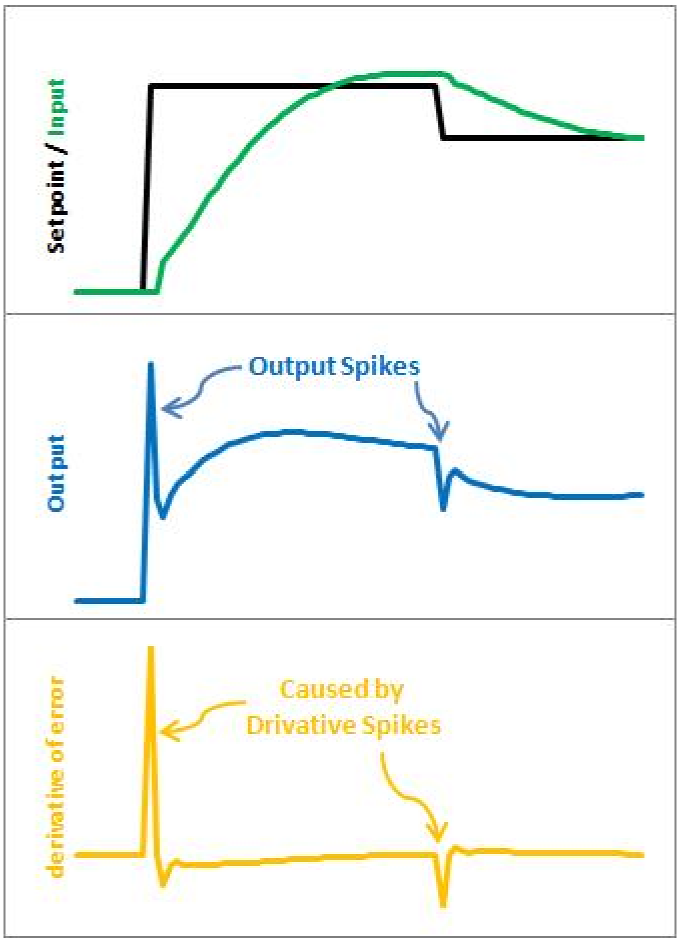
Derivative Kick 微分的毛刺

(This is Modification #2 in a larger series on writing a solid PID algorithm)

The Problem

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

通过修改对微分项进行弱化。目标是消除微分毛刺。（如下图）



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.

从上面的演示看到问题。当 **error=Setpoint-Input**，**setpoint** 改变之后，会导致出现一个极大的微分值（**error-error'**）**/timechange.**幸运的是，有简单解决办法。

The Solution

$$\frac{d\text{Error}}{dt} = \frac{d\text{Setpoint}}{dt} - \frac{d\text{Input}}{dt}$$

$$\text{When Setpoint is constant :}$$

$$\frac{d\text{Error}}{dt} = -\frac{d\text{Input}}{dt}$$

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"

除了 **setpoint** 的改变，微分项等于输入的负值。在公式里用 **–Kd *** 输入的微分 取代 **Kd***误差的微分。也就是微分测量的由来。

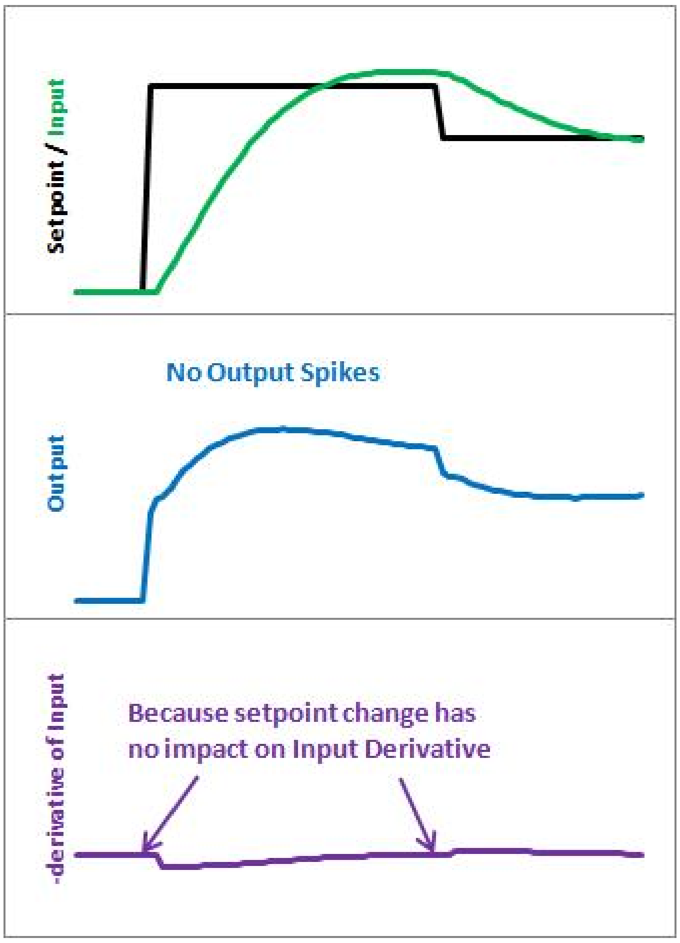
The Code

```
/*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

修改非常容易，我们用 **–dInput** 取代 **+dError**。 现在我们记忆，上一次的 **Input**，取代上一次的误差。

The Result 结果



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.

这就是我们修改的结果。注意 **inout** 似乎是一样的，所以我们得到了相似的性能，并且不会因为 **setpoint** 的变化导致 **output** 有巨大的纹波产生。

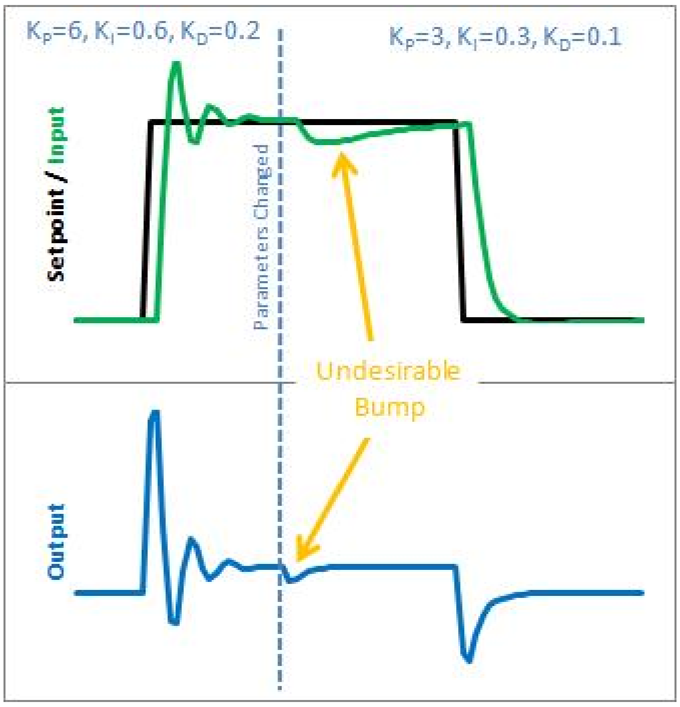
This may or may not be a big deal. It all depends on how sensitive your application is to output spikes. The way I see it though, it doesn't take any more work to do it without kicking so why not do things right?

这不是最大的问题。它取决于你的应用对纹波的敏感性。我想只要一点点的工作就能去掉纹波，为什么不那么做呢？

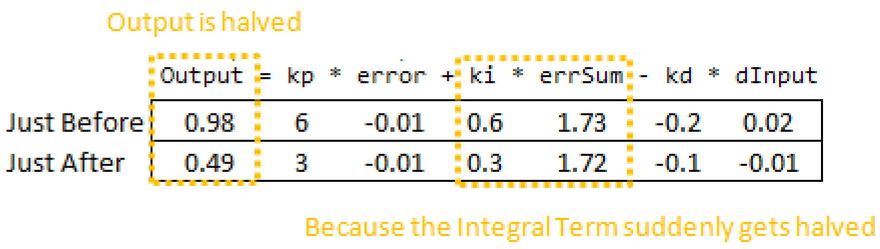
The Problem

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

因为输入参数的改变导致 **pid** 输出的变化。



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:



Output is halved

	Output =	kp *	error +	ki *	errSum -	kd *	dInput
Just Before	0.98	6	-0.01	0.6	1.73	-0.2	0.02
Just After	0.49	3	-0.01	0.3	1.72	-0.1	-0.01

Because the Integral Term suddenly gets halved

//以下文章的理解翻译：

这张表看出如果 **Ki** 的参数减为一半，**output** 也会减少一半输出。变化是巨大的。其实我们理想中 **setpoint=input** 的时候，**output** 因为没有变化，现在由于动态的修改了 **Ki** 的参数，就导致了问题的产生。其实要解决也很容易，因为 **I** 项是对之前所有无法的积分（也就是累加），如果动态的改变了 **Ki** 的值，会导致 **PID** 有大的变化。其实我们只要保持之前的积分值不变，之后的积分值直接乘以改变后的 **Ki** 后，就可以了。

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:

$$K_I \int e\,dt \approx K_{I_n}\left[e_n + e_{n-1} + \dots\right]$$

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?)

$$K_I \int e\,dt = \int K_I\, e\,dt$$

$$\int K_I\, e\,dt \approx K_{I_n} e_n + K_{I_{n-1}} e_{n-1} + \dots$$

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.

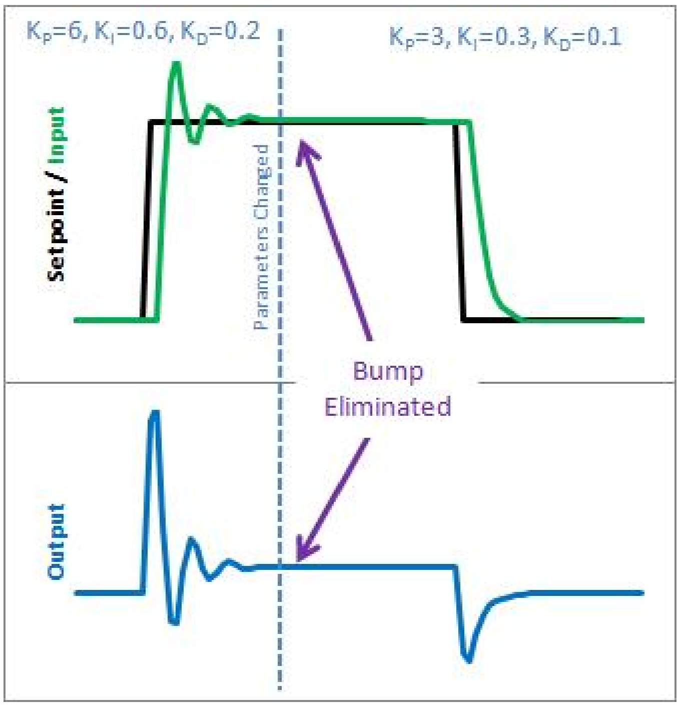
Now, we take the error and multiply it by whatever the Ki is at that time. We then store the sum of THAT. When the Ki changes, there's no bump because all the old Ki's are already "in the bank" so to speak. We get a smooth transfer with no additional math operations. It may make me a geek but I think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

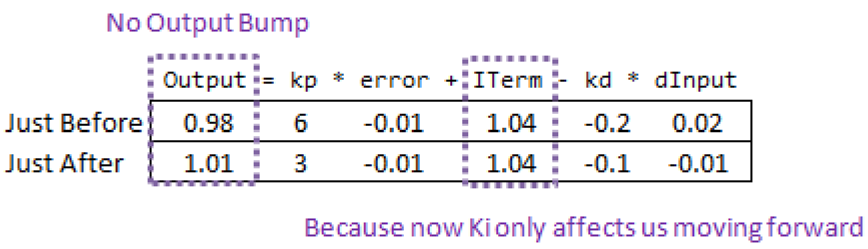
The Code

```
/*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);//这句话是精华
      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




No Output Bump
Because now Ki only affects us moving forward

	Output	= kp	* error	+ ITerm	- kd	* dInput
Just Before	0.98	6	-0.01	1.04	-0.2	0.02
Just After	1.01	3	-0.01	1.04	-0.1	-0.01

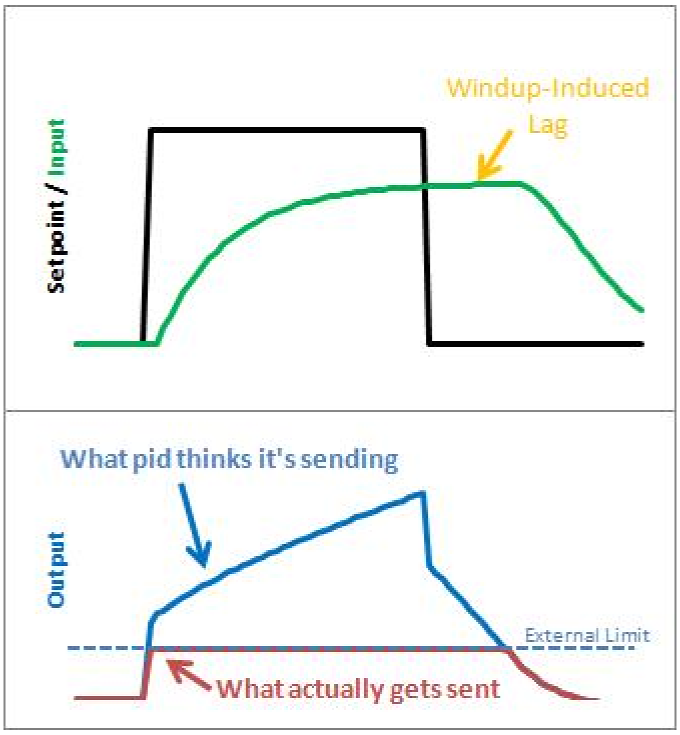
这里看到效果了吧。**Ki** 的变化没有对 **output** 有巨大的影响

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.

Improving the Beginner’s PID: Reset Windup

(This is Modification #4 in a larger series on writing a solid PID algorithm)

The Problem



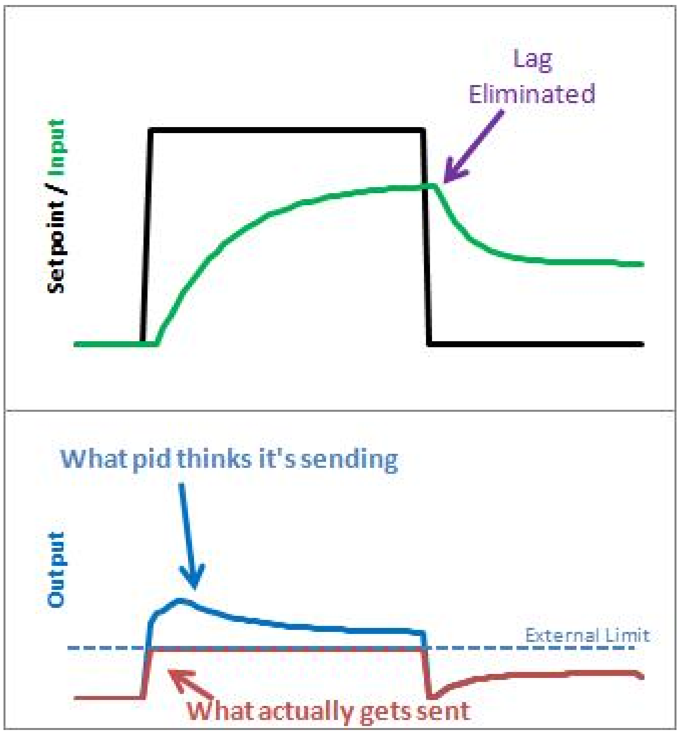
理解翻译：

因为应用的限制，导致 **PID** 的 **output** 输出超过了应用的范围。比如用 **DAC 8bit** 进行输出，输入最大的 **0-255**，而 **PID** 的 **output** 由于 **setpoint** 的改变，计算出来是 **300+**了，这时候问题就产生了。**PID** 以为已经输出了 **300+**，而实际输出的是 **255**，然后当 **setpoint** 突然变化成一个比较小的值。**PID** 会从 **300+**往下减，所以导致下降的时候输出有一个延迟，这个延迟只有下降到 **255** 的时候，**Input** 才有感觉。

Reset windup is a trap that probably claims more beginners than any other. It occurs when the PID thinks it can do something that it can’t. For example, the PWM output on an Arduino accepts values from 0-255. By default the PID doesn’t know this. If it thinks that 300-400-500 will work, it’s going to try those values expecting to get what it needs. Since in reality the value is clamped at 255 it’s just going to keep trying higher and higher numbers without getting anywhere.

The problem reveals itself in the form of weird lags. Above we can see that the output gets "wound up" WAY above the external limit. When the setpoint is dropped the output has to wind down before getting below that 255-line.

The Solution - Step 1



理解翻译，第一步，我们对 **PID** 输出的时候进行输出值的判断：

增加

```
if(Output > outMax) Output = outMax;
else if(Output < outMin) Output = outMin;
```

和

SetOutputLimits（）函数，PID 中 ITerm 还是在累加的，所以导致 PID 认为自己的计算还在增大。但是输出已经看起来很好了，并且下降趋势的输出还是有延迟的，改善不大。

There are several ways that windup can be mitigated, but the one that I chose was as follows: tell the PID what the output limits are. In the code below you'll see there's now a SetOuputLimits function. Once either limit is reached, the pid stops summing (integrating.) It knows there's nothing to be done; Since the output doesn't wind-up, we

get an immediate response when the setpoint drops into a range where we can do something.

The Solution - Step 2

Notice in the graph above though, that while we got rid that windup lag, we're not all the way there. There's still a difference between what the pid thinks it's sending, and what's being sent. Why? the Proportional Term and (to a lesser extent) the Derivative Term.

Even though the Integral Term has been safely clamped, P and D are still adding their two cents, yielding a result higher than the output limit. To my mind this is unacceptable. If the user calls a function called "SetOutputLimits" they've got to assume that that means "the output will stay within these values." So for Step 2, we make that a valid assumption. In addition to clamping the I-Term, we clamp the Output value so that it stays where we'd expect it.

(Note: You might ask why we need to clamp both. If we're going to do the output anyway, why clamp the Integral separately? If all we did was clamp the output, the Integral term would go back to growing and growing. Though the output would look nice during the step up, we'd see that telltale lag on the step down.)

理解翻译：

```
if(ITerm> outMax) ITerm= outMax;
else if(ITerm< outMin) ITerm= outMin;
```

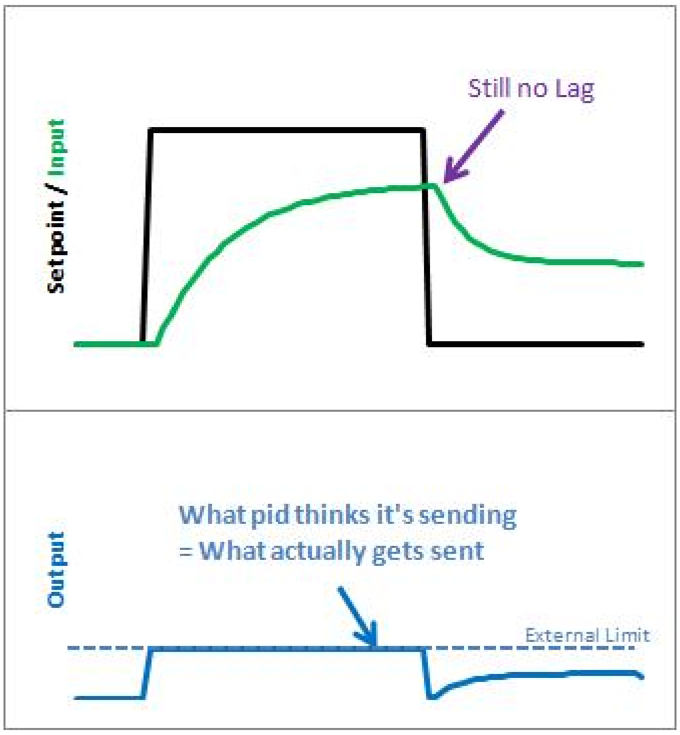
如果积分项设置在 **output** 之内，**PID** 很容易产生及时反映 **setpoint** 和 **outpout** 有关系的现象。这就导致了既没有延迟,（还是有很小的延迟的，可接受）**PID** 自己的计算还是知道自己的值在 **OUTPUT** 范围内的。

The Code

```
/*working variables*/
unsigned long lastTime;
double Input, Output, Setpoint;
double ITerm, lastInput;
double kp, ki, kd;
int SampleTime = 1000; //1 sec
double outMin, outMax;
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);
      if(ITerm> outMax) ITerm= outMax;
      else   if(ITerm<   outMin)   ITerm=
outMin;
      double dInput = (Input - lastInput);

      /*Compute PID Output*/
      Output = kp * error + ITerm- kd *
dInput;
      if(Output > outMax) Output = outMax;
      else if(Output < outMin) Output =
outMin;

      /*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
                      /
54  (double)SampleTime;
55        ki *= ratio;
56        kd /= ratio;
57        SampleTime       =       (unsigned
58  long)NewSampleTime;
59     }
60  }
61
62  void  SetOutputLimits(double Min,  double
63  Max)
    {
       if(Min > Max) return;
       outMin = Min;
       outMax = Max;

       if(Output > outMax) Output = outMax;
       else  if(Output  <  outMin)  Output  =
    outMin;

       if(ITerm> outMax) ITerm= outMax;
       else if(ITerm< outMin) ITerm= outMin;
    }
```

A new function was added to allow the user to specify the output limits [lines 52-63]. And these limits are used to clamp both the I-Term [17-18] and the Output [23-24]

The Result



As we can see, windup is eliminated. in addition, the output stays where we want it to. this means there's no need for external clamping of the output. if you want it to range from 23 to 167, you can set those as the Output Limits.

这里看到最上面的图和 **step1** 是一致的，下面那个图 **PID** 知道自己的值，其实是限制了 **Iterm** 值在范围内，从而限制 **PID** 的总和值。并且最后在判断一下 **PID** 的总值是在 **OUTPUT** 的可执行范围内的。

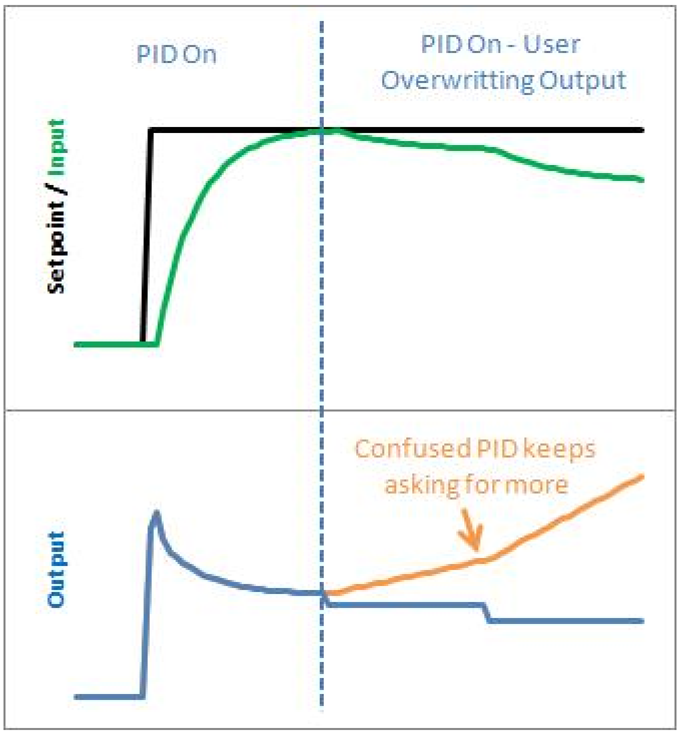
Improving the Beginner’s PID: On/Off

PID 的开和关

(This is Modification #5 in a larger series on writing a solid PID algorithm)

The Problem

As nice as it is to have a PID controller, sometimes you don′t care what it has to say.



Let′s say at some point in your program you want to force the output to a certain value (0 for example) you could certainly do this in the calling routine:

```
void loop()
{
Compute();
Output=0;
}
```

This way, no matter what the PID says, you just overwrite its value. This is a terrible idea in practice however. The PID will become very confused: "I keep moving the output, and nothing′s happening! What gives?! Let me move it some more." As a result, when you

stop over-writing the output and switch back to the PID, you will likely get a huge and immediate change in the output value.

理解翻译：上面的程序，你可以不去理会 **Compute()**计算出来的 **Output** 的值，而是直接对 **output** 赋值。这个方法非常糟糕，因为 **PID** 努力的去把 **Input** 的值和 **Setpoint** 的值进行比较，然后得出 **output** 的值，但是 **output** 又被你手动控制了。这是相当于 **PID** 是关闭的。当 **output** 值再次恢复 **PID** 公式接管的时候，会产生一个界跃，也就是 **output** 值会非常的大。

The Solution

The solution to this problem is to have a means to turn the PID off and on. The common terms for these states are "Manual" (I will adjust the value by hand) and "Automatic" (the PID will automatically adjust the output). Let's see how this is done in code:

The Code

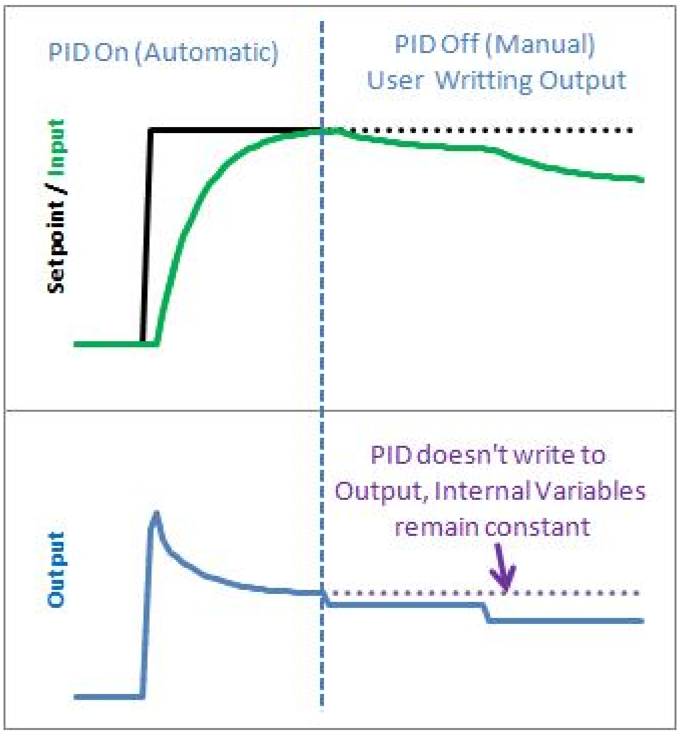
```
/*working variables*/
unsigned long lastTime;
double Input, Output, Setpoint;
double ITerm, lastInput;
double kp, ki, kd;
int SampleTime = 1000; //1 sec
double outMin, outMax;
bool inAuto = false;

#define MANUAL 0
#define AUTOMATIC 1

void Compute()
{
   if(!inAuto) return;
   unsigned long now = millis();
   int timeChange = (now - lastTime);
   if(timeChange>=SampleTime)
   {
      /*Compute all the working error
variables*/
      double error = Setpoint - Input;
      ITerm+= (ki * error);
      if(ITerm> outMax) ITerm= outMax;
      else if(ITerm< outMin) ITerm=
outMin;
      double dInput = (Input - lastInput);

      /*Compute PID Output*/
      Output = kp * error + ITerm- kd *
dInput;
      if(Output > outMax) Output = outMax;
      else if(Output < outMin) Output =
outMin;

      /*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;
   }
}

void  SetOutputLimits(double Min,  double
Max)
{
   if(Min > Max) return;
   outMin = Min;
```

```
      outMax = Max;

      if(Output > outMax) Output = outMax;
      else  if(Output  <  outMin)  Output  =
   outMin;

      if(ITerm> outMax) ITerm= outMax;
      else if(ITerm< outMin) ITerm= outMin;
   }

   void SetMode(int Mode)
   {
     inAuto = (Mode == AUTOMATIC);
   }
```

A fairly simple solution. If you're not in automatic mode, immediately leave the Compute function without adjusting the Output or any internal variables.

理解翻译：增加一个全局的开关，进入 **pid** 计算前，判断一下时候真正要计算，如果不要计算，也就是手动修改 **output** 的值。那就直接跳出 **pid** 计算。

The Result



It's true that you could achieve a similar effect by just not calling Compute from the calling routine, but this solution keeps the workings of the PID contained, which is kind

of what we need. By keeping things internal we can keep track of which mode were in, and more importantly it let′s us know when we change modes. That leads us to the next issue...

从 **on** 状态到 **off** 状态似乎没有什么问题了。但是从 **off** 状态到 **on** 状态时候就没有问题了呢？

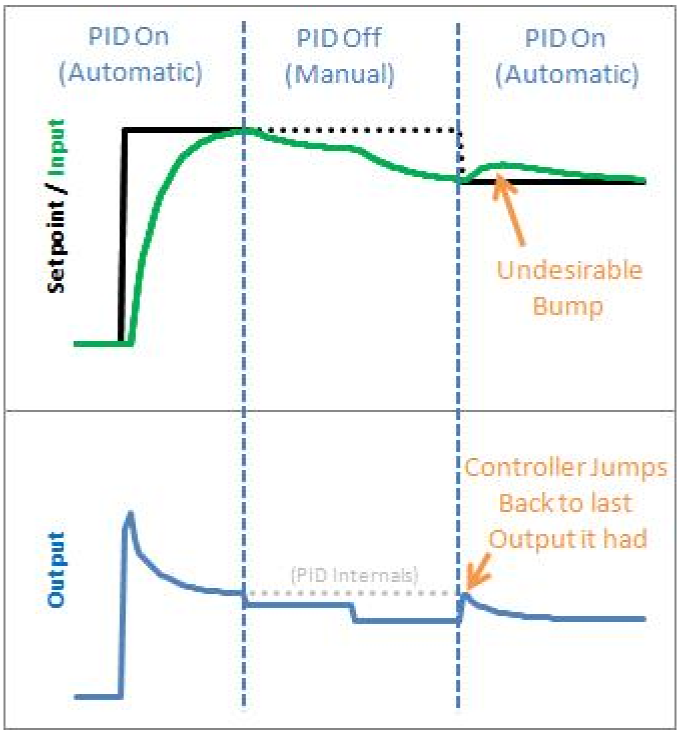
Improving the Beginner’s PID: Initialization

(This is Modification #6 in a larger series on writing a solid PID algorithm)

PID 初始化

The Problem

In the last section we implemented the ability to turn the PID off and on. We turned it off, but now let’s look at what happens when we turn it back on:



从图上看，**PID** 从关闭到打开，**OUTPUT** 会恢复到关闭前的状态值，我们的理想是 **OUTPUT** 不要有界跃，应该平滑。

Yikes! The PID jumps back to the last Output value it sent, then starts adjusting from there. This results in an Input bump that we’d rather not have.

The Solution

理解翻译：

解决的办法：

首先判断从 off 到 on:

```
  if(newAuto && !inAuto)
    {  /*we just went from manual to auto*/
        Initialize();
    }
```

然后进入 Initialize();

```
void Initialize()
{
   lastInput = Input;
   ITerm = Output;
   if(ITerm> outMax) ITerm= outMax;
   else if(ITerm< outMin) ITerm= outMin;
}
```

在 PID 初始化的时候主要是 2 个参数的初始化，一个是 LastInput，这个影响的是微分值，不会让微分变化太大。第二个是 ITerm 值，这个值在 PID 中数值最大，因为是一个积分值，所以要把 ITerm 等于 OUTPUT。这样 PID 计算出来就是一个在关闭到打开的时候 OUTPUT 的值附近产生一个 PID 计算的 OUTPUT 新值。

This one is pretty easy to fix. Since we now know when we're turning on (going from Manual to Automatic,) we just have to initialize things for a smooth transition. That means massaging the 2 stored working variables (ITerm & lastInput) to keep the output from jumping.

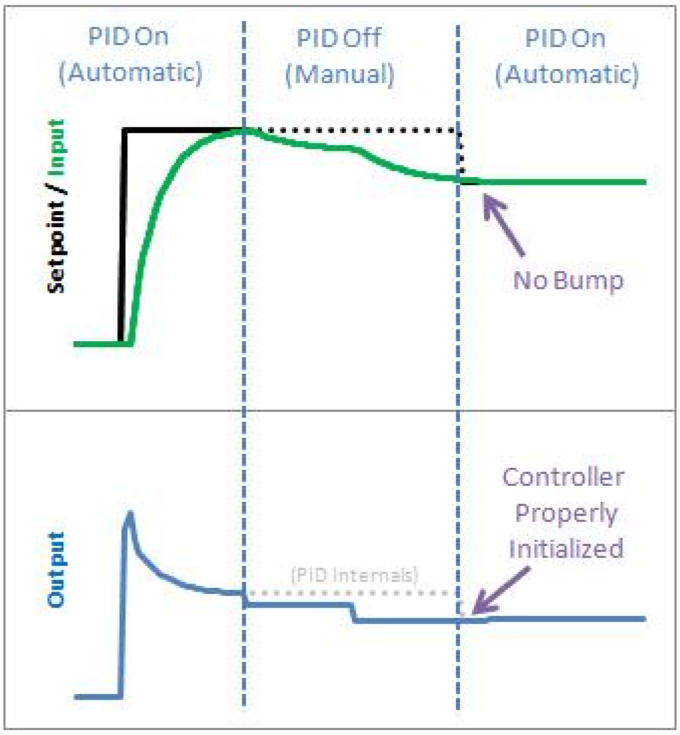
The Code

```
/*working variables*/
unsigned long lastTime;
double Input, Output, Setpoint;
double ITerm, lastInput;
double kp, ki, kd;
int SampleTime = 1000; //1 sec
double outMin, outMax;
bool inAuto = false;

#define MANUAL 0
#define AUTOMATIC 1

```

```
void Compute()
{
   if(!inAuto) return;
   unsigned long now = millis();
   int timeChange = (now - lastTime);
   if(timeChange>=SampleTime)
   {
      /*Compute all the working error
variables*/
      double error = Setpoint - Input;
      ITerm+= (ki * error);
      if(ITerm> outMax) ITerm= outMax;
      else if(ITerm< outMin) ITerm=
outMin;
      double dInput = (Input - lastInput);

      /*Compute PID Output*/
      Output = kp * error + ITerm- kd *
dInput;
      if(Output> outMax) Output = outMax;
      else if(Output < outMin) Output =
outMin;

      /*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;
   }
}

void  SetOutputLimits(double Min,  double
Max)
{
   if(Min > Max) return;
   outMin = Min;
   outMax = Max;

   if(Output > outMax) Output = outMax;
   else  if(Output  <  outMin)  Output  =
outMin;

   if(ITerm> outMax) ITerm= outMax;
   else if(ITerm< outMin) ITerm= outMin;
}

void SetMode(int Mode)
{
    bool newAuto = (Mode == AUTOMATIC);
    if(newAuto && !inAuto)
    {  /*we just went from manual to auto*/
        Initialize();
    }
    inAuto = newAuto;
}

void Initialize()
{
   lastInput = Input;
   ITerm = Output;
   if(ITerm> outMax) ITerm= outMax;
   else if(ITerm< outMin) ITerm= outMin;
}
```

We modified SetMode(…) to detect the transition from manual to automatic, and we added our initialization function. It sets ITerm=Output to take care of the integral term, and lastInput = Input to keep the derivative from spiking. The proportional term doesn′t rely on any information from the past, so it doesn′t need any initialization.

The Result



We see from the above graph that proper initialization results in a bumpless transfer from manual to automatic: exactly what we were after.

可以看到从 **off** 变成 **on** 之后，**output** 变为平滑了，没有毛刺了。

Improving the Beginner's PID: Direction

PID 输出 OUTPUT 的方向

(This is the last modification in a larger series on writing a solid PID algorithm)

The Problem

理解翻译：前面几章讲的，增大 OUTPUT 的值，INPUT 的值也是增大的。但是用在制冷系统里面增大 OUTPUT 的值，INPUT 的减小的。INPUT 就是温度传感器输出的值。解决办法很简单，Kp Ki Kd 都为负值。确保 3 个参数都是同样的符号的。（这个是非常重要的）

The processes the PID will be connected to fall into two groups: direct acting and reverse acting. All the examples I've shown so far have been direct acting. That is, an increase in the output causes an increase in the input. For reverse acting processes the opposite is true. In a refrigerator for example, an increase in cooling causes the temperature to go down. To make the beginner PID work with a reverse process, the signs of kp, ki, and kd all must be negative.

This isn't a problem per se, but the user must choose the correct sign, and make sure that all the parameters have the same sign.

The Solution

To make the process a little simpler, I require that kp, ki, and kd all be >=0. If the user is connected to a reverse process, they specify that separately using the SetControllerDirection function. this ensures that the parameters all have the same sign, and hopefully makes things more intuitive.

The Code

```
/*working variables*/
unsigned long lastTime;
double Input, Output, Setpoint;
double ITerm, lastInput;
double kp, ki, kd;
int SampleTime = 1000; //1 sec
double outMin, outMax;
bool inAuto = false;

#define MANUAL 0
```

```
#define AUTOMATIC 1

#define DIRECT 0
#define REVERSE 1
int controllerDirection = DIRECT;

void Compute()
{
   if(!inAuto) return;
   unsigned long now = millis();
   int timeChange = (now - lastTime);
   if(timeChange>=SampleTime)
   {
      /*Compute  all  the  working  error
variables*/
      double error = Setpoint - Input;
      ITerm+= (ki * error);
      if(ITerm > outMax) ITerm= outMax;
      else  if(ITerm  <  outMin)  ITerm=
outMin;
      double dInput = (Input - lastInput);

      /*Compute PID Output*/
      Output = kp * error + ITerm- kd *
dInput;
      if(Output > outMax) Output = outMax;
      else if(Output < outMin) Output =
outMin;

      /*Remember some variables for next
time*/
      lastInput = Input;
      lastTime = now;
   }
}

void  SetTunings(double  Kp,  double  Ki,
double Kd)
{
   if (Kp<0 || Ki<0|| Kd<0) return;

  double        SampleTimeInSec         =
((double)SampleTime)/1000;
   kp = Kp;
   ki = Ki * SampleTimeInSec;
   kd = Kd / SampleTimeInSec;

  if(controllerDirection ==REVERSE)
   {
      kp = (0 - kp);
      ki = (0 - ki);
      kd = (0 - kd);
   }
}

void SetSampleTime(int NewSampleTime)
{
   if (NewSampleTime > 0)
   {
      double          ratio           =
(double)NewSampleTime
                      /
(double)SampleTime;
      ki *= ratio;
      kd /= ratio;
      SampleTime      =      (unsigned
long)NewSampleTime;
   }
}

void SetOutputLimits(double Min, double
Max)
{
   if(Min > Max) return;
   outMin = Min;
   outMax = Max;

   if(Output > outMax) Output = outMax;
   else  if(Output  <  outMin)  Output  =
outMin;

   if(ITerm > outMax) ITerm= outMax;
   else if(ITerm < outMin) ITerm= outMin;
}

void SetMode(int Mode)
{
    bool newAuto = (Mode == AUTOMATIC);
    if(newAuto == !inAuto)
    {  /*we just went from manual to
auto*/
        Initialize();
    }
    inAuto = newAuto;
}

void Initialize()
{
   lastInput = Input;
   ITerm = Output;
   if(ITerm > outMax) ITerm= outMax;
   else if(ITerm < outMin) ITerm= outMin;
}

void SetControllerDirection(int
Direction)
{
   controllerDirection = Direction;
}
```

PID COMPLETE

PID 结束语：作者希望读到这篇文章的人能有所收获。

And that about wraps it up. We've turned "The Beginner's PID" into the most robust controller I know how to make at this time. For those readers that were looking for a detailed explanation of the PID Library, I hope you got what you came for. For those of you writing your own PID, I hope you were able to glean a few ideas that save you some cycles down the road.

Two Final Notes:

1. **If something in this series looks wrong please let me know. I may have missed something, or might just need to be clearer in my explanation. Either way I'd like to know.** 如果有不清楚的地方，让我知道。

2. **This is just a basic PID. There are many other issues that I intentionally left out in the name of simplicity. Off the top of my head: feed forward, reset tiebacks, integer math,**

different pid forms, using velocity instead of position. If there′s interest in having me explore these topics please let me know.

这是一片基础的 **PID** 文章，如果你有其他 **PID** 的算法，或者改进，请让我知道。