Arduino PID - Guía de uso de la librería

Traducción del trabajo de Brett Beauregard:
http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/

Licencia:

Moyano Jonathan Ezequiel [jonathan215.mza@gmail.com]

PID para principiantes, primer acercamiento:

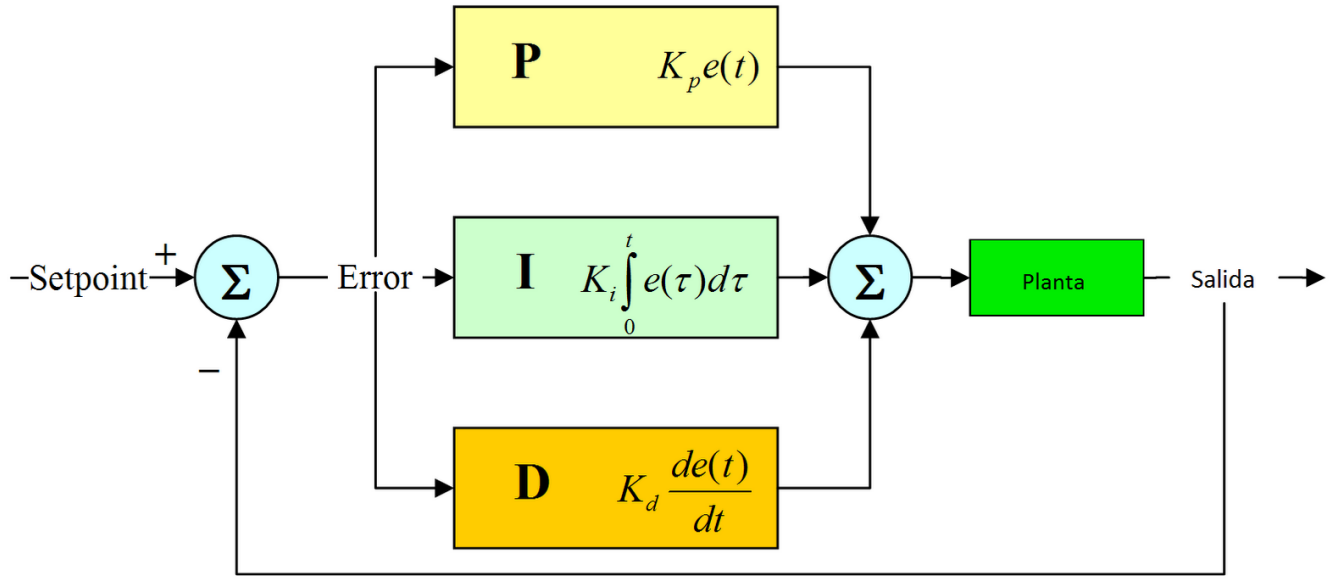
En esta introducción, veremos los parámetros básicos a tener en cuenta sobre el control propocional, integral , derivativo (PID); el objetivo de este tutorial no es introducirnos en los análisis teóricos del PID, sinó ver su aplicación en un sistema real, utilizando un microcontrolador programado en un lenguaje de alto nivel, como puede ser C.

La ecuación del PID:

De la documentación existente sobre sistemas de control, podemos destacar la siguiente ecuación.

$$u(t) = K_p e(t) + \frac{K_p}{T_i}\int_0^t e(t)dt + K_p T_d \frac{de(t)}{dt}$$

Para tener una idea más clara, recurrimos al siguiente diagrama



De la ecuación, podemos hacer las siguientes afirmaciones:

- **e(t)** es el error de la señal.
- **u(t)** salida del controlador y entrada de control al proceso.
- **Kp** es la ganancia proporcional.
- **Ti** es la constante de tiempo integral.
- **Td** es la constante de tiempo derivativa.

Del diagrama de flujo determinamos lo siguiente:

- El primer bloque de control (***proporcional***) consiste en el producto entre la señal de error y la constante proporcional, quedando un error en estado estacionario casi nulo.
- El segundo bloque de control (***integral***) tiene como propósito disminuir y eliminar el error en estado estacionario, provocado por el modo proporcional. El control integral actúa cuando hay una desviación entre la variable y el punto de consigna, integrando esta desviación en el tiempo y sumándola a la acción proporcional.
- El tercer bloque de control (***Derivativo***) considera la tendencia del error y permite una repercusión rápida de la variable después de presentarse una perturbación en el proceso.

Explicado lo anterior, tenemos el siguiente código:

```
/* Variables utilizadas en el controlador PID. */
unsigned long lastTime;
double Input, Output, Setpoint;
double errSum, lastErr;
double kp, ki, kd;

void Compute()
{
   /* Cuanto tiempo pasó desde el último cálculo. */
   unsigned long now = millis();
   double timeChange = (double)(now - lastTime);

   /* Calculamos todas las variables de error. */
   double error = Setpoint - Input;
   errSum += (error * timeChange);
   double dErr = (error - lastErr) / timeChange;
/* Calculamos la función de salida del PID. */
```

```
   Output = kp * error + ki * errSum + kd * dErr;

   /* Guardamos el valor de algunas variables para el próximo ciclo de cálculo. */
   lastErr = error;
   lastTime = now;
}

/* Establecemos los valores de las constantes para la sintonización. */
void SetTunings(double Kp, double Ki, double Kd)
{
   kp = Kp;
   ki = Ki;
   kd = Kd;
}
```

El programa anterior funciona correctamente, pero tiene limitaciones en cuanto a su aplicación a un sistema real. Para que se comporte como un PID de nivel industrial, hay que tener en cuenta otros parámetros; el algoritmo del PID funciona mejor si se ejecuta a intervalos regulares, si se incorpora el concepto del tiempo dentro del PID, se pueden llegar a simplificar los cálculos.

El problema:

Los PID principiantes, están diseñados para ejecutarse a periodos irregulares, esto puede traer 2 problemas:

- Se tiene un comportamiento inconsistente del PID, debido a que en ocaciones se lo ejecuta regularmente y a veces no.
- Hay que realizar operaciones matemáticas extras para calcular los términos correspondientes a la parte derivada e integral del PID, ya que ambos son dependientes del tiempo.

La solución:

Hay que asegurarse que la función que ejecuta el PID lo haga regularmente. Basado en un tiempo de ejecución predeterminado, el PID decide si debe hacer cálculos o retornar de la función. Una vez que nos aseguramos que el PID se ejecuta a intervalos regulares, los cálculos correspondientes a la parte derivada e integral se simplifican.

```
// Variables utilizadas en el controlador PID.
unsigned long lastTime;
double Input, Output, Setpoint;
double errSum, lastErr;
double kp, ki, kd;
int SampleTime = 1000; // Seteamos el tiempo de muestreo en 1 segundo.

void Compute()
{
   unsigned long now = millis();
   int timeChange = (now - lastTime);
   // Determina si hay que ejecutar el PID o retornar de la función.
   if(timeChange>=SampleTime)
   {
      // Calcula todas las variables de error.
      double error = Setpoint - Input;
      errSum += error;
      double dErr = (error - lastErr);

      // Calculamos la función de salida del PID.
      Output = kp * error + ki * errSum + kd * dErr;

      // Guardamos el valor de algunas variables para el próximo ciclo de cálculo.
      lastErr = error;
      lastTime = now;
   }
}

/* Establecemos los valores de las constantes para la sintonización.
   Debido a que ahora sabemos que el tiempo entre muestras es constante,
   no hace falta multiplicar una y otra vez por el cambio de tiempo; podemos
   ajustar las constantes Ki y Kd, obteniendose un resultado matemático equivalente
   pero más eficiente que en la primera versión de la función. */

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)
   {

/* si el usuario decide cambiar el tiempo de muestreo durante el funcionamiento, Ki y Kd tendrán
que ajustarse para reflejar este cambio. */

      double ratio  = (double)NewSampleTime / (double)SampleTime;
      ki *= ratio;
```

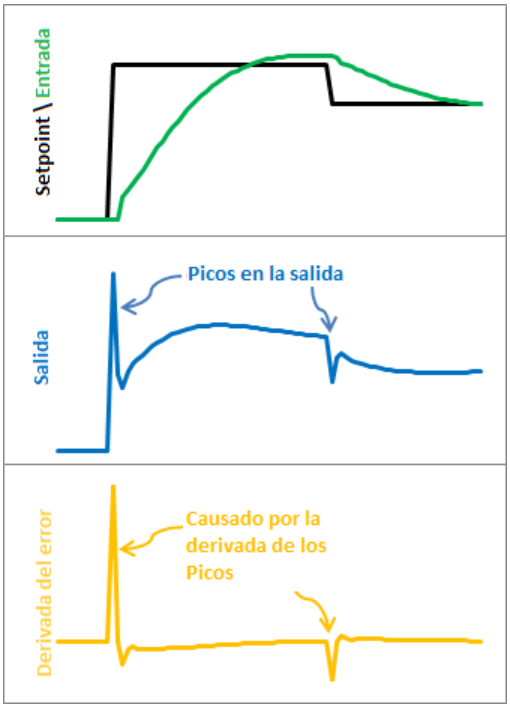
```
    kd /= ratio;
    SampleTime = (unsigned long)NewSampleTime; }}
```

Los resultados:

- Independientemente de cuán frecuente es llamada la función para calcular el PID, el algoritmo de control será evaluado a tiempos regulares.
- Debido la expresión (`int timeChange = (now - lastTime);`) no importa cuado `millis()` se hace cero nuevamente, ya que al tiempo actual, se le resta el tiempo transcurrido con anterioridad.
- Como el tiempo de muestreo ahora es constante, no necesitamos calcular permanentemente las constantes de sintonización. Con lo cuál nos ahorramos cálculos cada vez que procesamos el PID.
- Tener en cuenta que es posible mejorar la gestión de los tiempos de muestreos mediante interrupciones, pero queda a cargo del diseñador la implementación y prueba de este concepto.

Derivative Kick

Esta modificación que presentaremos a continuación, cambiará levemente el termino derivativo con el objetivo de eliminar el fenómeno "***Derivative Kick***". Este fenómeno, se produce por variaciones rápidas en la señal de referencia r(t), que se magnifican por la acción derivativa y se transforman en componentes transitorios de gran amplitud en la señal de control.



La imagen de arriba ilustra el problema. Siendo el *error = setpoint - entrada*, cualquier cambio en la consigna, causa un cambio instantáneo en el error; la derivada de este cambio es infinito (en la práctica, *dt* no es cero, igualmente, el valor termina siendo muy grande). Esto produce un sobrepico muy alto en la salida, que podemos corregir de una manera muy sencilla.

La solución:

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

Cuando el setpoint es constante

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

Resulta que la derivada del error es igual a la derivada negativa de la entrada, salvo cuando el setpoint está cambiando, esto acaba siendo una solución perfecta. En lugar de añadir (*Kd * error derivado*), restamos (*Kd * valor de entrada derivado*). Esto se conoce como el uso de "*Derivada de la medición*".

El código:

```
// Variables utilizadas en el controlador PID.

unsigned long lastTime;
double Input, Output, Setpoint;
double errSum, lastInput;
double kp, ki, kd;
int SampleTime = 1000; // Tiempo de muestreo de 1 segundo.

void Compute()
{
   unsigned long now = millis();
   int timeChange = (now - lastTime);
   if(timeChange>=SampleTime)
   {
      // Calcula todas las variables de errores.
      double error = Setpoint - Input;
      errSum += error;
      double dInput = (Input - lastInput);
```

```
      // Calculamos la función de salida del PID.
      Output = kp * error + ki * errSum - kd * dInput;

      // Guardamos el valor de algunas variables para el próximo ciclo de cálculo.
      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;
   }
}
```

Las modificaciones son bastante sencillas, estamos reemplazando la derivada positiva del error con la derivada negativa de la entrada. En vez de recordar el último valor del error, ahora recordamos el último valor que tomó la entrada.

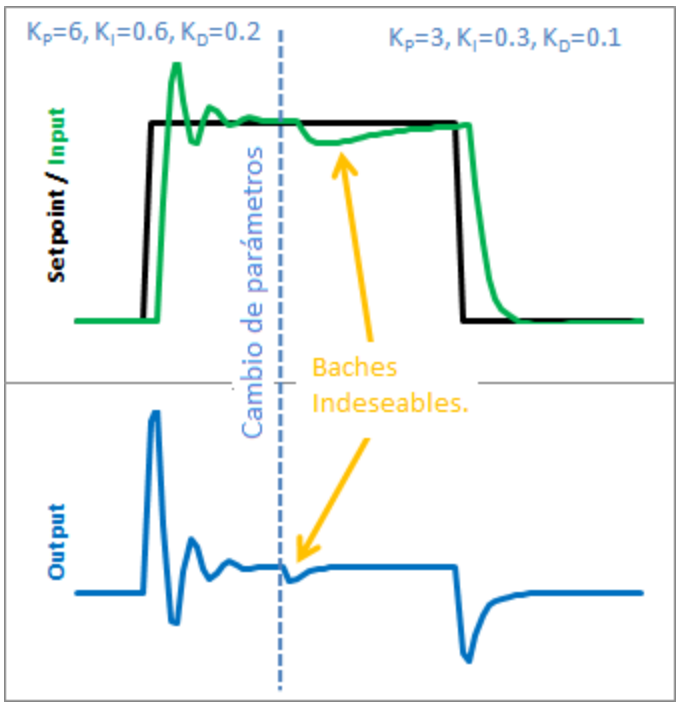
El resultado:

Podemos ver como los picos en la salida han sido eliminados. Este factor de corrección se aplica a sistemas muy sensibles a dichas variaciones; en un horno por ejemplo, donde la inercia térmica es muy grande, no le afectan en lo más mínimo dichos picos. Por lo tanto no sería necesario tener en cuenta esta corrección.

Cambios en la sintonización

El problema:

La posibilidad de cambiar la sintonización del PID, mientras el sistema está corriendo, es la característica más respetable del algoritmo del sistema de control.



Los PID principiantes, de hecho, actúan de manera errática si queremos setear los valores de la sintonización, mientras el sistema está corriendo. Veamos por que. Aquí se muetra el estado del PID antes y después de que los parámetros han cambiado.

La salida se reduce a la mitad

	Output	= kp	* error	+ ki	* errSum	- kd	* dInput
Justo Antes	0.98	6	-0.01	0.6	1.73	-0.2	0.02
Justo Después	0.49	3	-0.01	0.3	1.72	-0.1	-0.01

Debido a que el término integral rápidamente se ha reducido a la mitad.

De inmediato podemos ver que el culpable de este bache en la señal de salida es el término integral; es el único término que cámbia drásticamente cuando la señal de sintonización se modifica. Esto sucede debido a la interpretación de la integral.

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

Esta interpretación funciona bien hasta que Ki cambia. De repente, la suma de todos los errores se multiplica con el valor de Ki, esto no es lo que necesitamos. Nosotros solo queremos que afecte a los valores que estén por delante. Por ejemplo: Si nosotros modificamos Ki, en un tiempo t=5s. Necesitamos que el impacto de este cambio solo afecte a valores de Ki que se modifican en un tiempo mayor a t=5s.

La solución:

La solución a este error no queda muy elegante, pero consiste en reescalar la suma del error, doblando el valor de Ki o cortando la suma de los errores a la mitad. Esto quita el bache del termino integral, solucionando el problema.

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

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

En lugar de tener el término Ki fuera de la integral, lo introducimos dentro del cálculo. Al parecer, no hemos realizado nada extraño, pero en la práctica está

acción resulta en una grán diferencia en la función de salida del PID.

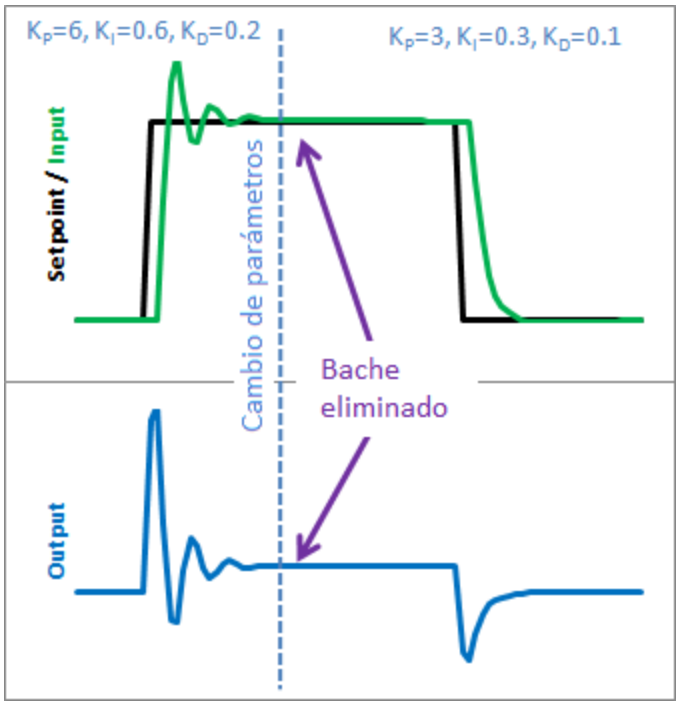
Ahora tomamos el error y lo multiplicamos por el valor de Ki en ese momento, luego almacenamos la suma de los diferentes errores multiplicados por la constante Ki. Esto resulta en una función de salida, suave y sin sobresaltos, con la ventaja de no tener que utilizar matemática adicional para ello.

```
// Variables utilizadas en el controlador PID.
unsigned long lastTime;
double Input, Output, Setpoint;
double ITerm, lastInput;
double kp, ki, kd;
int SampleTime = 1000; // Tiempo de muestreo: 1 segundo.

void Compute()
{
   unsigned long now = millis();
   int timeChange = (now - lastTime);
   if(timeChange>=SampleTime)
   {
      // Calcula todos los errores.
      double error = Setpoint - Input;
      ITerm += (ki * error);
      double dInput = (Input - lastInput);

      // Calculamos la función de salida del PID.
      Output = kp * error + ITerm - kd * dInput;

      // Guardamos el valor de algunas variables para el próximo recálculo.
      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;
```

```
    }
}
```

Reemplazamos la variable errSuma, por una variable compuesta llamada Iterm. Suma *Ki*error* en lugar de solamente el error. Por último, como el cálculo de Ki está embebido en el término integral, se elimina de la ecuación principal del PID.

El resultado:



No hay bache en la salida.

	Output	= kp * error +	ITerm	- kd * dInput	
Justo Antes	0.98	6 -0.01	1.04	-0.2	0.02
Justo Después	1.01	3 -0.01	1.04	-0.1	-0.01

Con las modificaciones hechas, los cambios en la sintonización del término integral no afectan al rendimiento general de nuestro sistema, ya que tiene en cuenta la modificación en cada instancia de error, sumandose al total.

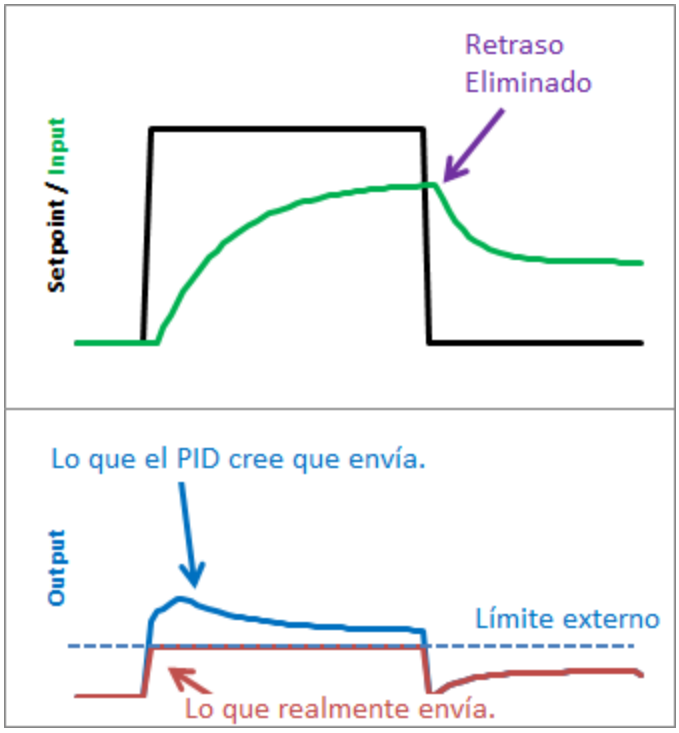
Reset WindUp

El problema:

El efecto windup aparece al arrancar el sistema o en cualquier otra situación, donde aparece un error muy grande durante un tiempo prolongado. Esto hará que el término integral aumente para reducir el error. Pero si nuestro actuador es limitado, con esto me refiero que la tensión que podemos aplicarle esta entre 0 y 5V (0 a 255 , pwm de 8 bits), se saturará, pero el termino integral seguirá creciendo. Cuando el error se reduce, la parte integral también comenzará a reducirse, pero desde un valor muy alto, llevando mucho tiempo hasta que logre la estabilidad, generando fluctuaciones exageradamente grandes.

El problema se manifiesta en forma de retrasos extraños. En la imagen podemos ver que el valor de la salida, está muy por encima del límite. Cuando el valor del setpoint cae por debajo de un valor determinado, el valor de salida decrece por debajo de la línea límite de 255 (5v).

La solución - Paso 1:



Hay varias formas para mitigar el efecto del WindUp, pero la elegida es la siguiente: decirle al PID cuáles son los límites de salida. En el código de abajo, veremos que ahora hay una función SetOuputLimits. Una vez que ya se alcanza el límite, el PID detiene el funcionamiento del término integral.

La solución - Paso 2:

Observe en el gráfico anterior, si bien nos libramos del retraso inducido por el WindUp, no hemos resuelto todo el problema. Todavía hay una diferencia, entre lo que el pid piensa que está enviando, y lo que está enviando. ¿Por qué? Veamos el término proporcional y (en menor medida) el término derivativo.

Aunque el término integral ha sido acotado de forma segura, el termino Proporcional y Derivativo están añadiendo pequeños valores adicionales, dando un resultado superior al límite de salida. Esto es inaceptable. Si el usuario llama a

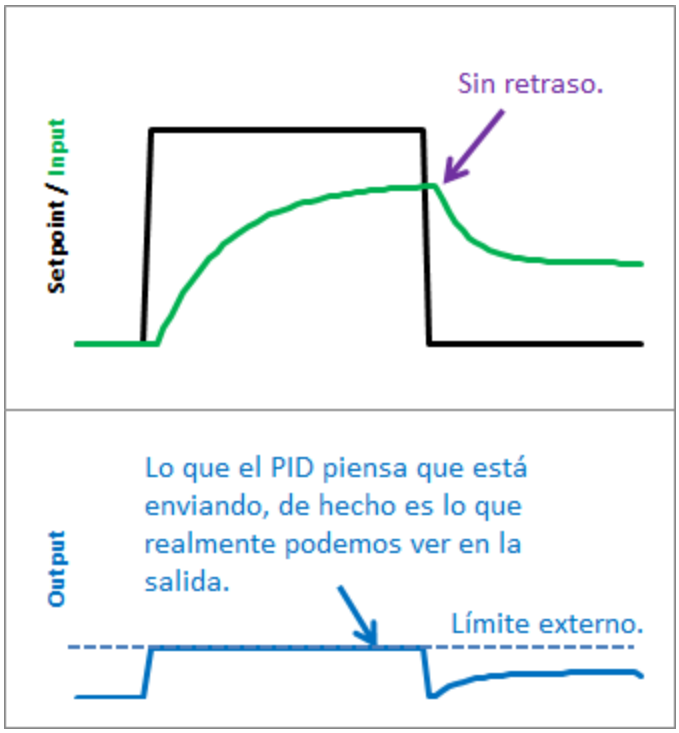
la función "SetOutputLimits" tiene que asumir que eso significa "la salida se mantendrá dentro de estos valores." Así que en el paso 2, hacemos una suposición válida. Además de la restricción del término Integral, hay que acotar el valor de salida para que se mantenga dentro de los límites.

Uno se preguntará, por que acotamos el termino integral y la salida. Esto se debe a lo siguiente: Por más que pongamos límites al valor que puede tomar la salida, el término integral seguiría creciendo, introduciendo errores en la salida.

El código:

```
// Variables de trabajo.
unsigned long lastTime;
double Input, Output, Setpoint;
double ITerm, lastInput;
double kp, ki, kd;
int SampleTime = 1000; // Tiempo de muestreo de 1 segundo.
double outMin, outMax;
void Compute()
{
   unsigned long now = millis();
   int timeChange = (now - lastTime);
   if(timeChange>=SampleTime)
   {
      // Calcula todos los errores.
      double error = Setpoint - Input;
      ITerm+= (ki * error);
      if(ITerm> outMax) ITerm= outMax;
      else if(ITerm< outMin) ITerm= outMin;
      double dInput = (Input - lastInput);

      // Calculamos la función de salida del PID.
      Output = kp * error + ITerm- kd * dInput;
      if(Output > outMax) Output = outMax;
      else if(Output < outMin) Output = outMin;

      // Guardamos el valor de algunas variables para el próximo recálculo.
      lastInput = Input;
      lastTime = now;
   }
}

void SetTunings(double Kp, double Ki, double Kd)
{
  double SampleTimeInSec = ((double)SampleTime)/1000;
   kp = Kp;
```

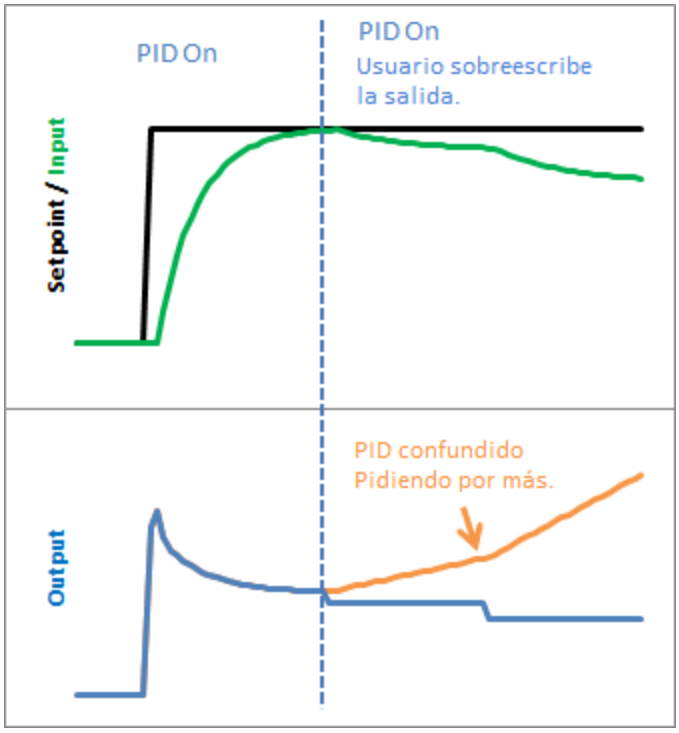
```
   ki = Ki * SampleTimeInSec;
   kd = Kd / SampleTimeInSec;
}

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;
}
```



Como podemos ver el fenómeno del WindUp es eliminado. Además, podemos ver que la salida permanece dentro del rango que deseamos. Esto significa que podemos configurar el rango de valores máximos y mínimos que necesitamos en la salida.

PID: On/Off

El problema:



Digamos que en algún momento del programa deseamos forzar la salida a un valor determinado (0 por ejemplo), usando la siguiente rutina:

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

De esta manera no importa el valor de salida que haya computado el PID, nosotros simplemente determinamos su valor manualmente. Esto en la práctica es erróneo ya que introducirá errores en el PID: Dirá, yo estoy variando la función de salida, pero en realidad no pasa nada. Como resultado, cuando pongamos nuevamente el PID en funcionamiento, tendremos un cambio brusco y repentino en el valor de la función de salida.

La solución:

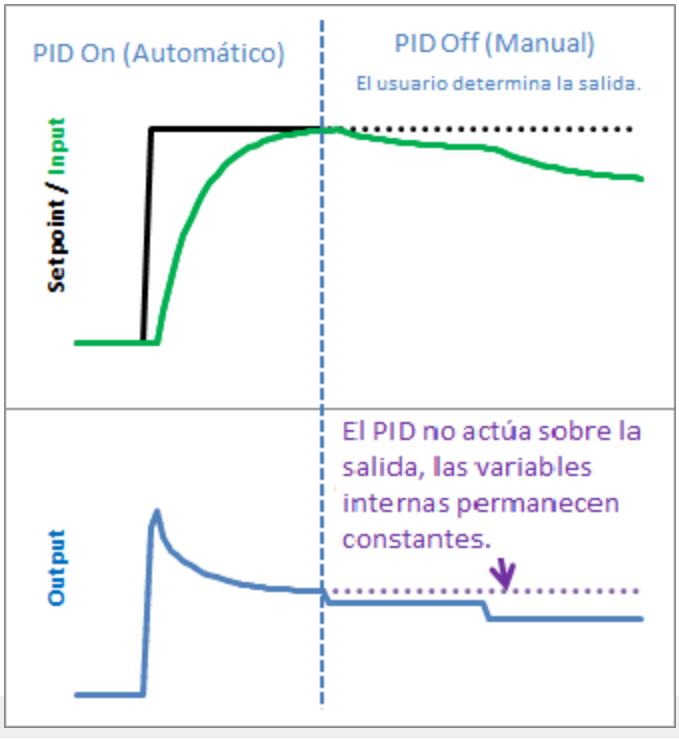
La solución a este problema es tener un medio para encender o apagar el PID de vez en cuando. Los términos comunes para estos estados son "**Manual**" (*ajustar el valor de la salida manualmente*) y "**Automatic**" (*el PID ajusta automáticamente la salida*). Vamos a ver cómo se hace esto en el código:

```
// Variables de trabajo.
unsigned long lastTime;
double Input, Output, Setpoint;
double ITerm, lastInput;
double kp, ki, kd;
int SampleTime = 1000; // Tiempo de muestreo 1 segundo.
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)
   {
      // Calculamos todos los errores.
      double error = Setpoint - Input;
      ITerm+= (ki * error);
      if(ITerm> outMax) ITerm= outMax;
      else if(ITerm< outMin) ITerm= outMin;
      double dInput = (Input - lastInput);

      // Calculamos la función de salida del PID.
      Output = kp * error + ITerm- kd * dInput;
      if(Output > outMax) Output = outMax;
      else if(Output < outMin) Output = outMin;
```

```
      // Guardamos el valor de algunas variables para el próximo recálculo.
      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);
}
```

Una solución bastante simple. Si no está en modo automático, sale inmediatamente de la función de cómputo del PID, sin ajustar la salida ni las variables internas del mismo.

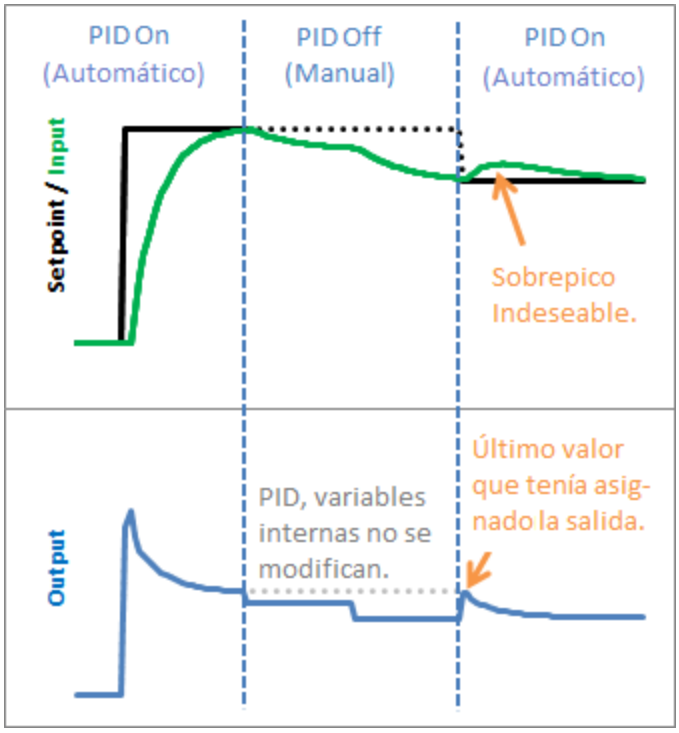
El resultado:



Ciertamente podríamos conseguir un efecto similar sin llamar a la función que calcula el PID, pero esta solución mantiene las variables del PID contenidas. Al mantener el valor de dichas variables, podemos hacer un seguimiento de los valores de las mismas, y lo más importante, vamos a saber cuando podemos cambiar los modos.

PID: Inicialización

Anteriormente habíamos implementado la posibilidad de encender o apagar el PID de vez en cuando. Ahora vamos a ver lo que pasa cuando volvemos a encenderlo:



Aquí tenemos un problema, el PID entrega a la salida el última valor computado, luego comienza a corregir a partir de ahí. Esto resulta en un sobrepico en la entrada que es preferible no tener.

La solución:

Esto es bastante fácil de solucionar. Ahora sabemos que al pasar de manual a automático, sólo tenemos que inicializar los parámetros para una transición sin problemas. Esto significa, inicializar el valor de la entrada con el último valor almacenado, e inicializar el término integral con el último valor que tomó la salida, para evitar los sobrepicos en la salida.

El código:

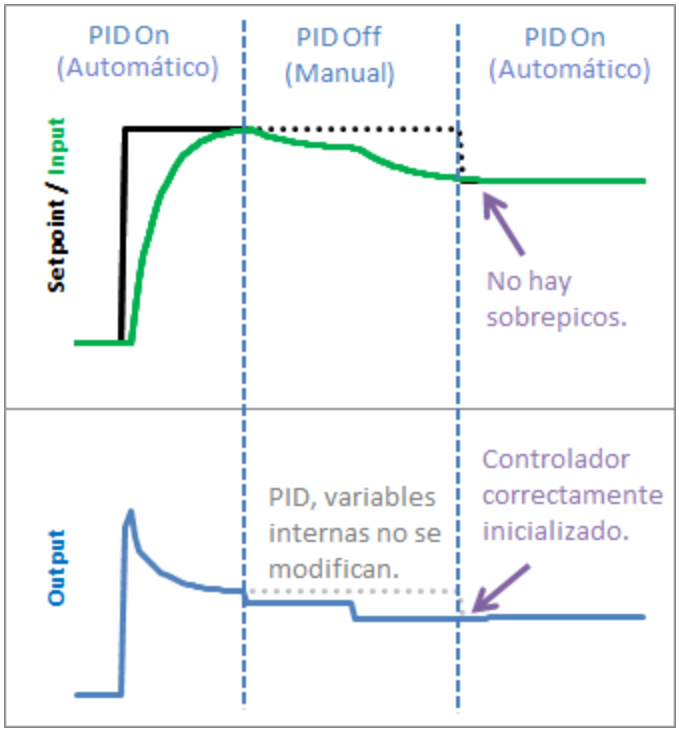
```
// Variables de trabajo.
unsigned long lastTime;
```

```
double Input, Output, Setpoint;
double ITerm, lastInput;
double kp, ki, kd;
int SampleTime = 1000; // Tiempo de muestreo 1 segundo.
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)
   {
      // Calculamos todos los errores.
      double error = Setpoint - Input;
      ITerm+= (ki * error);
      if(ITerm> outMax) ITerm= outMax;
      else if(ITerm< outMin) ITerm= outMin;
      double dInput = (Input - lastInput);

      // Calculamos la función de salida del PID.
      Output = kp * error + ITerm- kd * dInput;
      if(Output > outMax) Output = outMax;
      else if(Output < outMin) Output = outMin;

      // Guardamos el valor de algunas variables para el próximo recálculo.
      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)
    {  // Para cambiar de manual a automático, inicializamos algunos parámetros.
        Initialize();
    }
    inAuto = newAuto;
}

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

Hemos modificado setMode () para detectar el paso de manual a automático y hemos añadido nuestra función de inicialización. En él se establecen (`iTerm = salida`) cuidar de que el término integral, y (`LastInput = Entrada`) para mantener la derivada de la adición. El término proporcional no se basa en la información del pasado, por lo que no necesita ningún tipo de inicialización.

El resultado:



Vemos en el gráfico anterior que una inicialización adecuada, da como resultado, una transferencia de manual a automático sin perturbaciones: exactamente lo que estábamos buscando.

PID: Dirección

El problema:

Los procesos a los cuáles un PID estará enlazado, se dividen 2 grupos: de acción directa y de acción inversa. Todos los ejemplos vistos hasta el momento han sido de acción directa, por lo tanto, un incremento en la entrada, da como resultado un incremento en la salida. En el caso de los procesos de acción reversa, es todo lo contrario.

En un refrigerador, por ejemplo, un aumento en la acción de enfriamiento, causa una disminución de la temperatura. Para que el PID funcione en un proceso de acción inversa, los signos de Kp, Ki, y Kd deben ser negativos.

Esto no es un problema por si mismo, pero el usuario debe elegir el signo correcto, y asegúrese de que todos los parámetros tengan el mismo signo.

La solución:

Para hacer el proceso un poco más simple, se requiere que los parámetros Kp, Ki, y kd sean > = 0. Si el usuario está trabajando en un proceso de acción inversa, se especifica por separado, utilizando la función SetControllerDirection. esto asegura que los parámetros tienen el mismo signo.

El código:

```
// Variables de trabajo.
unsigned long lastTime;
double Input, Output, Setpoint;
double ITerm, lastInput;
double kp, ki, kd;
int SampleTime = 1000; // Tiempo de muestreo 1 segundo.
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)
   {
      // Calculamos todos los errores.
      double error = Setpoint - Input;
      ITerm+= (ki * error);
      if(ITerm> outMax) ITerm= outMax;
      else if(ITerm< outMin) ITerm= outMin;
      double dInput = (Input - lastInput);

      // Calculamos la función de salida del PID.
      Output = kp * error + ITerm- kd * dInput;
      if(Output > outMax) Output = outMax;
      else if(Output < outMin) Output = outMin;
```

```
      // Guardamos el valor de algunas variables para el próximo recálculo.
      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)
    {  // Para cambiar de manual a automático, inicializamos algunos parámetros.
        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;
}
```