Un Control PID para robots con LEGO® MINDSTORMS (2ª parte)

Os traemos la segunda y última parte del tutorial sobre MINDSTORMS. Esperamos que os animéis a mandarnos más sugerencias.

Texto y gráficos por J. Sluka

Traducción y adaptación por Jetro

Añadir la "I" al controlador: El controlador PI ("I": ¿qué has hecho por mi últimamente?)

Para mejorar la respuesta del controlador P añadiremos un nuevo factor a la ecuación. Este término se llama la **integral**, la "I" en PID. Las integrales son una parte muy importante de las matemáticas avanzadas, pero la parte que necesitamos es, afortunadamente, bastante sencilla.

La integral es la suma acumulada del error.

Si, es así de sencillo. Hay algunos pequeños asuntos que nos saltaremos de momento.

Cada vez que leemos el sensor de luz y calculamos el **error** añadiremos ese **error** a la variable que llamaremos **integral** (listo ¿verdad?).

integral = integral + error

Esa ecuación puede parecer algo extraña, y lo es. No está escrita como una ecuación matemática, sino que es una forma usada comúnmente en programación para sumar una serie de valores. Matemáticamente no tiene ningún sentido. En programación el signo 'igual a' tiene un significado un tanto diferente que en las matemáticas. (Emplearé la misma fuente tipo máquina de escribir que he usado para el pseudo código para resaltar que se trata de una forma de programar y no de matemáticas). El "=" significa haz las mates de la derecha y guarda el resultado en la variable que se indica a la izquierda. Queremos que el ordenador tome el valor antiguo de **integral**, y añada el **error** para luego guardar el resultado nuevamente en **integral**.

A continuación, al igual que hicimos con la **"P"**, multiplicaremos la **integral** por una constante proporcional, es decir, otra **K**. Ya que esta constante proporcional va con la integral la llamaremos **Ki**. Al igual que la P (proporcional), multiplicaremos la integral por una constante (**Ki**) para conseguir una corrección. Para nuestro robot siguelíneas es una adición a la variable **Giro**.

Giro = Kp*(error) + Ki*(integral)

Esta es la ecuación básica de un controlador PI. **Giro** es la corrección para los motores. El término proporcional es **Kp*(error)** y el término integral es **Ki*(integral)**.

¿Exactamente qué hace el término integral? Si el **error** mantiene el mismo signo durante varios bucles la **integral** crece y crece. Por ejemplo, si comprobamos el sensor de luz y calculamos que el **error** es 1, poco después lo comprobamos de nuevo y el **error** es 2, y la siguiente vez el **error** es 2 otra vez, entonces la **integral** será 1+2+2=5. La **integral** es 5 pero el **error** en este paso en particular es de sólo 2. La **integral** puede ser un importante factor en la corrección, pero tarda un tiempo hasta que se acumula y empieza a contribuir.

Otra cosa que hace la integral es eliminar pequeños errores. Si el sensor de luz de nuestro siguelíneas está muy cerca del borde de la línea, pero no exactamente encima, entonces el **error** será pequeño y solo hace falta una pequeña corrección. Tal vez puedas corregir ese pequeño **error** modificando **Kp** en el término proporcional, pero a menudo eso resulta en un robot que oscila (continuamente hace zig-zag). La **integral** es perfecta para arreglar pequeños errores. Como la **integral** suma los **errores**, varios pequeños **errores** sucesivos acabarán aumentando la **integral** hasta el punto de hacer la diferencia.

Se puede considerar la **integral** como la "memoria" del controlador. La **integral** es la historia acumulada del **error** y proporciona al controlador un método de subsanarlos si persisten en el tiempo.

Algunos detalles sobre la integral

Sí, la integral tiene más detalles. Afortunadamente no son demasiado dolorosos.

Pasé por alto un asunto menor (Vale, no es del todo menor, pero haremos que lo sea): el tiempo. La integral es en realidad la suma de **error***(delta time). Delta time (**dT**) es el tiempo transcurrido entre la última vez que leímos el sensor de luz y la lectura más reciente;

integral = integral + error*(dT)

Así que cada vez que sumamos algo a la integral deberíamos añadir el **error** multiplicado por **dT**. Es bastante sencillo hacer que el robot mida **dT**. Simplemente leeríamos un reloj cada vez que hacemos una lectura con el sensor de luz. Si restamos la última hora leída de la actual tendremos el tiempo transcurrido desde la última lectura de **dT**. (Hay mejores maneras de hacer esto, pero me las saltaré porque no hacen falta.) Pero ¿no sería agradable si no tuviera que medir **dT** y hacer la multiplicación?. Bueno, ¿y si **dT** siempre es igual? Cada vez que sumamos a la **integral** tenemos el mismo término **dT** . De modo que podemos eliminar el factor **dT** de **error*(dT)** y seguir sumando como hacíamos antes;

integral = integral + error

Solo si queremos hacer otro cálculo con la **integral** realmente tenemos que multiplicar por **dT**. Espera, hay más...

Podemos hacer más aún para esconder el término tiempo. El término integral en la ecuación del controlador PI es **Ki*(integral)*dT**. Pero **Ki** es un parámetro que necesitamos ajustar (al igual que **Kp**) así que ¿por qué no simplemente sustituir **Ki*dT** por un nuevo **Ki**? El nuevo **Ki** es diferente del original pero ya que no conocemos ninguno de los dos da igual cual usamos o cómo lo llamamos. Independientemente de cómo lo llamamos o qué representa, tenemos que averiguar el valor correcto principalmente por prueba y error.

De modo que hemos eliminado por completo el factor tiempo del término integral con la restricción de que todos los pasos de tiempo, **dT**, son iguales (aproximadamente).

La integral tiene memoria de elefante

Un último detalle acerca de la **integral**. Normalmente la **integral** solo se puede mover hacia cero, donde no contribuye nada al controlador, añadiendo valores de **error** del signo contrario de la mayoría que hemos ido añadiendo a la **integral**. Por ejemplo, si durante varios ciclos del bucle los **errores** son 1,2,2,3,2,1 eso suma hasta tener una **integral** de 11. Pero el **error** en la última lectura es de solo 1, mucho menos que la **integral** en ese momento. La única manera de que la **integral** vaya hacia cero es conseguir una serie de **errores** negativos para compensar la serie de errores positivos anterior y así bajar la **integral**. Por ejemplo, si los siguientes errores son -2,-2,-3, la integral bajará de 11 a 4 y aún harían falta más errores negativos para bajar la **integral** a cero. Además, la **integral** quiere que el **error** total esté uniformemente distribuido entre valores positivos y negativos.

Si sucede algo que empuja nuestro siguelíneas hacia la izquierda del borde de la línea, la **integral** no solo querrá volver al borde de la línea sino que la sobrepasará hacia la derecha por la misma distancia que la desviación hacia la izquierda. Así que la integral crece si hay grandes errores que persisten durante un tiempo. Esto puede causar problemas con el controlador que tiene una **integral**. A veces esta tendencia de la **integral** de sobrecompensar al intentar corregir el **error** es lo suficientemente problemática para que el programador tenga que hacer algo con la **integral** para evitar estos problemas. Si la 'crecida' de la **integral** causa problemas, las dos soluciones comunes son (1) poner la **integral** a cero, es decir, asignar un valor cero a la **integral** cada vez que el valor del **error** es cero o el **error** cambia de signo. (2) "Amortiguar" la **integral**, multiplicando la integral acumulada por un factor inferior a 1 al calcular la nueva **integral**. Por ejemplo:

integral = (2/3)*integral + error

Esto reduce el valor anterior de la integral a un tercio cada vez que se ejecuta el bucle. Si piensas en la integral como la "memoria" del controlador, esta amortiguación es como obligarlo a olvidar cosas que pasaron hace "mucho" tiempo.

Pseudo código para el controlador PI

Para añadir la integral al controlador tenemos que añadir una nueva variable para **Ki** y una para la **integral** misma. Y no olvides que estamos multiplicando las Ks por 100 para paliar el problema de las matemáticas integrales.

```
Kp = 1000                    ! RECUERDA que estamos usando Kp*100 así en realidad esto es 10 !
Ki = 100                     ! RECUERDA que estamos usando Ki100 así en realidad esto es 1 !
offset = 45                  ! Inicializar las variables
Tp = 50
integral = 0                 ! el lugar donde almacenaremos nuestra integral
Bucle infinito
  ValorLuz = leer sensor de luz   ! cual es la lectura actual de luz
  error = ValorLuz - offset       ! calcular el error restando el offset
  integral = integral + error     ! la nueva integral
  Giro = Kp*error + Ki*integral   ! la "P" y la "I"
```

```
  Giro = Giro/100                                  ! RECUERDA deshacer el efecto de la multiplicación por 100 en Kp !
  potenciaA = Tp + Giro                            ! el nivel de potencia para el motor A
  potenciaC = Tp - Giro                            ! el nivel de potencia para el motor C
  MOTOR A dirección=hacia delante potencia=potenciaA   ! ejecutar el comando en el bloque MOTOR
  MOTOR C dirección=hacia delante potencia=potenciaC   ! ejecutar el comando en el bloque MOTOR
fin de bucle infinito                              ! bucle terminado: vuelve al inicio del bucle y ejecútalo otra vez
```

Añadir la "D" al controlador: El controlador PID completo ("D": ¿qué pasará ahora?)

Nuestro controlador ahora contiene un término proporcional (P) que intenta corregir el **error** actual y un término **integral** (I) que intenta corregir **errores** pasados. ¿Hay una manera de que el controlador mire hacia delante en el tiempo para intentar corregir un **error** que aún no haya sucedido?

Si, y la solución es otro concepto de las matemáticas avanzadas llamado la **derivada**. Aaaa, ahí está la "D" de PID. Al igual que la **integral**, la **derivada** puede conllevar cálculos matemáticos bastante serios. Afortunadamente, lo que necesitamos para el PID es bastante sencillo.

Podemos ver el futuro suponiendo que el siguiente cambio en el **error** será igual que el último.

Eso significa que esperamos que el siguiente **error** sea igual al **error** actual más el cambio en el **error** entre las dos lecturas de sensor precedentes. El cambio en el **error** entre dos lecturas consecutivas se llama la **derivada**. La **derivada** es igual a la inclinación de la línea.

Puede parecer complejo de calcular, pero realmente no es tan difícil. Un conjunto de datos de muestra nos ayudarán a ver cómo funciona. Supongamos que el **error** actual es 2 y que el **error** anterior esa 5. ¿Qué error predecimos para el siguiente?. Bueno, El cambio en el error es la **derivada** a saber;

(el **error** actual) - (el **error** anterior)

Con nuestros números sale 2 - 5 = -3. La **derivada** actual, por lo tanto es -3. Para usar la **derivada** para predecir el siguiente **error** usaríamos

(siguiente error) = (el **error** actual) + (la **derivada** actual)

Con nuestros números sale 2 + (-3) = -1. De modo que predecimos que el siguiente **error** será -1. En la práctica no llegamos tan lejos como para predecir el siguiente **error**. En vez de eso usamos la **derivada** directamente en el cálculo del controlador.

El término D, al igual que el término I, debería incluir un factor tiempo y el término D "oficial" es:

Kd(derivada)/(dT)

Al igual que con los términos **proporcional** e **integral** tenemos que multiplicar por una constante. Ya que esta constante va con la **derivada** se llama **Kd**. Fíjate también en que dividimos la derivada entre **dT** mientras que la **integral** se multiplica por **dT**. No te preocupes demasiado por el porqué de esto ya que aplicaremos el mismo método para eliminar **dT** de la **derivada** que empleamos con la **integral**. La división **Kd/dT** es una constante si **dT** es igual para cada bucle. Así que podemos cambiar **Kd/dT** por otra **Kd**. Como esta K, al igual que las anteriores Ks es desconocida y se determina mediante prueba y error, no importa si es **Kd/dT** o simplemente un nuevo valor para **Kd**.

Ahora podemos formular la ecuación completa para un controlador PID:

Giro = Kp*(error) + Ki*(integral) + Kd*(derivada)

Es evidente que "predecir el futuro" es algo muy práctico, pero ¿exactamente cómo nos ayuda esto? ¿Y que grado de fiabilidad tiene esa predicción?

Si el **error** actual es peor que el **error** anterior, entonces el término D intenta corregir el **error**. Si el **error** actual es mejor que el **error** anterior, entonces el término D intenta evitar que el controlador corrija el **error**. Es este segundo caso el que nos interesa especialmente. Si el **error** se acerca a cero nos estamos acercando al punto donde queremos dejar de corregir el error. Como el sistema probablemente tarde un tiempo en responder a los cambios en la potencia de los motores queremos empezar a reducir esa potencia antes de que el error haya llegado a cero; de lo contrario nos pasaremos. Cuando lo miras así pudiera parecer que el cálculo de D debería ser más complejo, pero no es así. De lo único que tienes que preocuparte es de hacer la resta en el orden correcto. El orden correcto para este tipo de cosas es "actual" menos "anterior". Así que para calcular la **derivada** tomamos el **error** actual y restamos el **error** anterior.

Pseudo código para el controlador PID

Para añadir la derivada al controlador tenemos que añadir una nueva variable para **Kd** y una variable para recordar el último **error**. Y no olvides que estamos multiplicando las Ks por 100 para paliar el problema de las matemáticas integrales.

```
Kp = 1000                          ! RECUERDA que estamos usando Kp*100 así en realidad esto es 10 !
Ki = 100                           ! RECUERDA que estamos usando Ki100 así en realidad esto es 1 !
Kd = 10000                         ! RECUERDA que estamos usando Kd*100 así en realidad esto es 100 !
offset = 45                        ! Inicializar las variables
Tp = 50
integral = 0                       ! el lugar donde almacenaremos nuestra integral
ultimoError = 0                    ! el lugar donde almacenaremos el valor del último error
derivada = 0                       ! el lugar donde almacenaremos la derivada
Bucle infinito
  ValorLuz = leer sensor de luz    ! cual es el valor de luz actual?
  error = ValorLuz - offset        ! calcular el error restando el offset
  integral = integral + error      ! calcular la integral
  derivada = error - ultimoError   ! calcular la derivada
  Giro = Kp*error + Ki*integral + Kd*derivada          ! La "P", la "I" y la "D"
 Giro = Giro/100                   ! RECUERDA deshacer el efecto de la multiplicación por 100 en Kp, Ki y Kd!
  potenciaA = Tp + Giro            ! el nivel de potencia para el motor A
  potenciaC = Tp - Giro            ! el nivel de potencia para el motor C
  MOTOR A dirección=hacia delante potencia=potenciaA    ! ejecutar el comando en el bloque MOTOR
  MOTOR C dirección=hacia delante potencia=potenciaC    ! igual que el otro motor, pero usando el otro nivel de potencia
  ultimoError = error              ! guardar el error actual para usarlo como último Error en el siguiente bucle
fin de bucle infinito              ! bucle terminado: vuelve al inicio del bucle y ejecútalo otra vez
```

Ahora tenemos el pseudo código para el controlador PID para un robot siguelíneas completo. Ahora empieza lo complicado: "ajustar" el PID. El ajuste es el proceso para encontrar los mejores valores para **Kp**, **Ki** y **Kd** o al menos valores que funcionan.

Ajustar un controlador PID sin matemáticas complejas (Aún así hay que hacer algunas mates)

Los más inteligentes ya han averiguado cómo ajustar un controlador PID. Como yo no soy tan inteligente como ellos, usaré lo que he aprendido. Resulta que medir algunos parámetros del sistema te permite calcular unos valores "bastante buenos" para **Kp**, **Ki** y **Kd**. No importa mucho exactamente qué sistema se intenta controlar, los cálculos de ajuste casi siempre funcionan bastante bien. Hay varias técnicas para calcular Ks, uno de los cuales se llama el *"Método Ziegler–Nichols "* que es lo que usaremos. Si buscas en google encontrarás muchas páginas web que explican esta técnica con todo detalle. La versión que usaré viene casi directamente de la página wiki sobre controladores PID (el mismo método se encuentra en muchas otras páginas). Solo haré un pequeño cambio introduciendo el tiempo del bucle (**dT**) en los cálculos de la tabla más abajo.

Para ajustar el controlador PID hay que seguir lo siguientes pasos.

1. Da un valor de cero a Ki y Kd lo que en efecto los 'apaga' de modo que el controlador funcionará como un simple controlador P.

2. Elije un valor pequeño para Tp . Para nuestros motores, 25 sería un buen comienzo.

3. Elije un valor "razonable" para Tp . ¿Qué es un valor "razonable"?
    1. Yo tomo el valor máximo que queremos enviar a los motores (100) y lo divido entre el máximo error útil. Para nuestro siguelíneas hemos supuesto un error máximo de 5 así que el valor de Kp que estimaremos es 100/5=20. Cuando el error es +5, la potencia del motor variará 100 unidades. Cuando el error es cero, la potencia del motor se quedará en el valor de Tp.
    2. También puedes poner un Kp de 1 (o de 100) y observar qué sucede.
    3. Si el código requiere valores de K 100 veces mayor a lo real tendrás que tenerlo en cuenta. 1 se convierte en 100, 20 en 2000, 100 en 10000.

4. Pon el robot en marcha y observa lo que hace. Si no puede seguir la línea y se aleja, aumenta Kp. Si oscila demasiado reduce Kp. Sigue variando Kp hasta que encuentres un valor que te proporciona una oscilación notable pero no exagerada. A este valor de Kp lo llamaremos "Kc" ("ganancia crítica" en la literatura sobre PID).

5. Usando el valor Kc para Kp, deja que el robot siga una línea y determina la velocidad de la oscilación. Esto puede ser difícil, pero afortunadamente no hace falta que la medición sea muy exacta. El período de oscilación (Pc) es el tiempo que el robot necesita para oscilar de un extremo de la línea al otro y volver. Para un robot LEGO típico Pc estará entre 0.5 y 2 segundos.

6. También necesitas saber la velocidad con la que el robot ejecuta los bucles. Haz que el robot ejecute el bucle un número determinado de buces (por ejemplo 10.000) y cronometra el tiempo que tarda en hacerlo (o haz que el mismo robot lo grabe y muestre en la pantalla). El tiempo del bucle (dT) es el tiempo medido dividido entre el número de bucles. Para un controlador PID completo, escrito en NXT-G, sin más parafernalia, dT estará entre 0,015 y 0,020 segundos por bucle.

7. Usa la tabla de abajo para calcular los valores de Kp, Ki, y Kc. Si quieres un controlador P usa la línea marcada con P para calcular el valor de Kp "correcto" (Ki' y Kd' serán cero). Si quieres un controlador PI usa la siguiente línea. Para un controlador PID completo usa la última línea.

8. Si tu código necesita Ks multiplicado por 100 no tienes que hacer esa multiplicación para estos valores. Ese factor 100 ya está

contemplado en el valor de Kp = Kc que determinaste antes.

9. Pon el robot en marcha y observa lo que hace.

10. Ajusta los valores de Kp, Ki y Kd para optimizar el robot.. Puedes empezar con cambios bastante grandes, digamos de un 30%, para luego ir haciéndolos más pequeños hasta llegar a los valores óptimos (o al menos aceptables).

11. Una vez que tengas unas buenas Ks intenta aumentar el valor Tp que controla la velocidad del robot si va recto.

12. Reajusta las Ks o incluso vuelve al paso 1 y repite el proceso entero para el nuevo valor de Tp.

13. Sigue repitiendo esto hasta que el comportamiento del robot sea aceptable.

Método Ziegler–Nichols dando valores para K' (el tiempo de bucle es considerado constante e igual a dT)

Tipo de control	Kp	Ki'	Kd'	
P	0,50Kc	0	0	
PI	0,45Kc	1,2KpdT/ Pc	0	
PID	0,60Kc	2KpdT/ Pc	KpPc / (8dT)	

Las apóstrofes sobre **Ki**' y **Kd**' simplemente sirven para recordar que se calculan suponiendo un que **dT** es una constante y que ha sido integrado en los valores **K**.
No encontré las ecuaciones para el controlador PD. Si alguien las conoce por favor mándame un email.

Estos son los valores que medí para mi robot [*] Kc era 300 y cuando **Kp**=**Kc** el robot oscilaba en aproximadamente 0,8 segundos, de modo que **Pc** es 0,8. Medí Pc contando en alto cada vez que el robot se giraba en una determinada dirección. Luego comparé mi percepción de lo rápido que contaba con "1-segundo, 2 -segundos, 3-segundos..-.". No es exactamente "ingeniería de alta precisión" pero funcionó bastante bien así que lo llamaremos "ingeniería práctica". El tiempo del bucle, **dT**, es 0,014 segundos/bucle, averiguado haciendo un programa con 10.000 bucles y haciendo que el NXT muestre el tiempo de ejecución. Aplicando la tabla del controlador PID obtenemos

Kp = (0.60)(Kc) = (0.60)(300) = 180
Ki = 2(**Kp**)(**dT**) / (**Pc**) = 2(180)(0.014) / (0.8) = 6.3 (lo que redondeamos a 6)
Kd = (**Kp**)(**Pc**) / ((8)(**dT**)) = (180)(0.8) / ((8)(0.014)) = 1286

Después de más prueba y error hallé los valores 220, 7, y 500 para **Kp**, **Ki** y **Kd** respectivamente. Recuerda que todos mis Ks son en realidad 100x el valor real de modo que los valores reales son 2,2, 0,07 y 5.

Cómo los cambios en Kp, Ki y Kd afectan al robot

La tabla y método descritos anteriormente son un buen punto de partida para optimizar tu PID. A veces ayuda tener una idea más clara de lo que sucede al aumentar (o disminuir) alguna de las tres Ks. La tabla siguiente está disponible en muchas webs. Esta en particular es de la Wiki sobre control PID.

Efectos de aumentar los parámetros

Parámetro	Tiempo de establecimiento	Sobreimpulso	Tiempo de estabilización	Error en equilibrio
Kp	Disminuye	Aumenta	Cambio pequeño	Disminuye
Ki	Disminuye	Aumenta	Aumenta	Elimina
Kd	Sin definir (pequeño aumento o disminución)	Disminuye	Disminuye	Ninguno

El "tiempo de establecimiento" define lo rápido que el robot intenta corregir un error. En nuestro ejemplo es lo que tarda el robot en intentar volver al borde de la línea después de alejarse. El tiempo de establecimiento es controlado principalmente por **Kp**. Un **Kp** mayor hará que el robot intente volver más rápido y reducirá el tiempo de establecimiento. Si **Kp** es demasiado grande el robot tendrá un sobreimpulso.

El "sobreimpulso" es cuanto el robot se tiende a desviar de la línea al responder ante un error. Por ejemplo, si el sobreimpulso es pequeño, el robot no pasará a la derecha de línea al intentar un error hacia la izquierda. Si el sobreimpulso es grande, el robot sobrepasará ampliamente el borde de la línea al corregir un error. El sobreimpulso es controlado en gran medida por **Kd** pero se ve muy afectado por **Ki** y **Kp**. Normalmente hay que aumentar **Kd** para corregir un exceso de sobreimpulso. ¿Recuerdas nuestro siguelíneas inicial, el que solamente podía girar a la derecha o la izquierda? Ese siguelíneas tiene mucho sobreimpulso. De hecho es lo único que hace.

El "tiempo de estabilización" es cuanto tarda el robot en recuperarse ante un cambio grande. En el caso de nuestro siguelíneas hay un gran cambio cuando el robot encuentra un giro. A medida que el robot responde a la curva corregirá el **error** y luego

presentará un sobreimpulso de una determinada medida. Luego tiene que corregir ese sobreimpulso y podría tener un sobreimpulso hacia el lado contrario. Luego tiene que corregir ese sobreimpulso... bueno, seguro que te haces la idea. Como el robot está respondiendo a un error tenderá a oscilar alrededor de la posición deseada. El “tiempo de estabilización” es cuento dura esa oscilación hasta volver a cero. Tanto **Ki** como **Kd** influyen mucho en el tiempo de estabilización. A mayor **Ki** mayor tiempo de estabilización. A mayor **Kd** menor tiempo de estabilización.

“Error en equilibrio” es el error que queda cuando el sistema funciona sin alteraciones externas. Para nuestro siguelíneas sería la distancia desde la línea al seguir una línea recta. A menudo los controles P y PD acaban teniendo este error. Se puede reducir aumentando **Kp** pero eso puede resultar en que el robot oscile. Incluir una I y reducir **Ki** a menudo arreglará un control P o PD con un error en equilibrio constante. (Suponiendo que no importe mucho el pequeño error restante a medida que el robot sigue la línea. Simplemente significa que hay una pequeña desviación hacia un lado u otro).

¿Qué tal funciona?

Hay un breve vídeo de un robot siguelíneas LEGO MINDSTORMS básico, siguiendo la línea en la hoja de pruebas que viene con el set. La calidad del vídeo no es muy buena.
El sensor de luz está a aproximadamente 1,3 cm del suelo y hacia un lado de la línea central del robot. La **Tp** (potencia de objetivo) se fijó en 70%. El robot avanza un promedio unos 20cm por segundo. Es un siguielíneas del lado izquierdo y sigue el interior de la oval. Es algo más difícil seguir el interior que el exterior de la línea.

MPEG4 - MP4 (644KB) QuickTime - MOV (972KB)

El siguelíneas parece funcionar bastante bien. Si ves el vídeo observarás que el robot se menea algo al salir de las curvas. Eso es el PID que oscila un poco. Cuando se mueve hacia la cámara puedes ver la mancha roja en el suelo que proviene del LED sensor de luz. Parece que sigue el borde de la línea bastante bien.

El control PID básico funcionará para muchos diferentes problemas de control, y por supuesto se puede implementar como control P o PI en vez de PID. Tendrías que buscar una nueva definición del error y habría que ajustar el PID para esa tarea específica.
#