A PID Controller For LEGO® MINDSTORMS Robots (Part 2)

We bring you the second and last part of the MINDSTORMS tutorial. We hope you feel encouraged to send us more suggestions

Text and pictures by J. Sluka

Adding "I" To The Controller: The PI Controller ("I": what have you done for me lately?)

To improve the response of our P controller we will add a new term to the equation. This term is called the **integral**, the "I" in PID. Integrals are a very important part of advanced mathematics, fortunately the part we need is pretty straight forward.

The integral is the running sum of the error.

Yep, it's that simple. There are a few subtle issues we'll skip for the moment.

Each time we read the light sensor and calculate an **error** we will add that **error** to a variable we will call **integral** (clever eh?).

integral = integral + error

That equation might look a little odd, and it is. It isn't written as a mathematical statement, it is written in a common form used in programming to add up a series of values. Mathematically it doesn't make any sense. In computer programming the equals sign has a somewhat different meaning than in math. (I'll use the same typewriter font I used for the pseudo code examples to highlight that it is a programming form and not a proper mathematical form.) The "=" means do the math on the right and save the result in the variable named on the left. We want the computer to get the old value of **integral**, add the **error** to it then save the result back in **integral**.

Next, just like the P term, we will multiply the **integral** by a proportionality constant, that's another K. Since this proportionality constant goes with the integral term we will call it **Ki**. Just like the proportional term we multiply the integral by the constant (**Ki**) to get a correction. For our line following robot it is an addition to our **Turn** variable.

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

The above is the basic equation for a PI controller. **Turn** is our correction for the motors. The proportional term is **Kp*(error)** and the integral term is **Ki*(integral)**.

What exactly does the integral term do for us? If the **error** keeps the same sign for several loops the **integral** grows bigger and bigger. For example, if we check the light sensor and calculate that the **error** is 1, then a short time later we check the sensor again and the the **error** is 2, then the next time the error is 2 again, then the **integral** will be 1+2+2=5. The **integral** is 5 but the **error** at this particular step is only 2. The integral can be a large factor in the correction but it usually takes a while for the **integral** to build up to the point where it starts to contribute.

Another thing that the integral does is it helps remove small errors. If in our line follower the light sensor is pretty close to the line's edge, but not exactly on it, then the **error** will be small and it will only take a small correction to fix. You might be able to fix that small **error** by changing **Kp** in the proportional term but that will often lead to a robot that oscillates (wobbles back and forth). The **integral** term is perfect for fixing small errors. Since the **integral** adds up the **errors**, several consecutive small **errors** eventually makes the **integral** big enough to make a difference.

One way to think about the **integral** term is that it is the controller's "memory". The **integral** is the cumulative history of the **error** and gives the controller a method to fix errors that persist for a long time.

Some subtle issues with the integral

Yep, the integral has more detail. Fortunately they aren't too painful.

I glossed over a minor issue (OK, it really isn't minor but we are going to make it so), the time. The integral is really the sum of the **error***(delta time). Delta time (**dT**) is the time between the last time we checked the sensor and the time of the most recent check of the sensor;

integral = integral + error*(dT)

So every time we add to **integral** the thing we should add is the **error** times the **dT**. It is fairly easy to have the robot measure the **dT**. We would just read a timer each time we read the light sensor. If we subtract the last time from the current time we get the time since the last reading **dT**. (There are better ways to do this but I'll skip'm since they are not needed.) But wouldn't it be nice if didn't have to measure the **dT** and do the multiplication? Well, what if the **dT** is always the same? Every time we add to **integral** we have that same **dT** term. So we can take that factor of **dT** out of **error*(dT)** and just do the summing the way we did before;

integral = integral + error

Only when we want to do another calculation with **integral** do we actually need to multiply by **dT**. But wait there's more...

We can do even more to hide the time term. The integral term in the PI controller equation is **Ki*(integral)*dT**. But **Ki** is a parameter that we have to fine tune (just like **Kp**) so why not just replace the **Ki*dT** part with a new **Ki**? The new **Ki** is different from the original but since we don't know either one it doesn't really matter which one we use or what we call it. No matter what we call it or what it represents we still have to find the correct value largely by trial and error.

So we have completely removed the time element for the integral term with the restriction that all the times steps, **dT**s, are the same (or about the same).

The integral has a memory like an elephant

One last detail should be mentioned about the **integral**. Usually the **integral** can only be moved towards zero, where it doesn't contribute anything to the controller, by having **error** values added that are the opposite sign of most of the ones that we have already collected in **integral**. For example, if over several cycles through the loop the **errors** are 1,2,2,3,2,1 that adds up to an **integral** of 11. But the **error** at the last data point is only 1, which is much smaller than the **integral** at that point. The only way for the **integral** to move towards zero is to get a string of negative **errors** to counter balance the earlier string of positive **errors** to "wind down" the **integral**. For example, if the next few errors are -2,-2,-3 then the integral will drop from 11 to 4 and we would still need more negative **errors** to get the **integral** down to zero. In addition, the **integral** wants the total **error** to be evenly distributed between positive and negative errors.

If something happens that pushes our line following robot to the left of the line's edge the **integral** term not only wants to get back to the line's edge it also wants to overshoot the edge to the right by as much as the original disturbance was the left. So the integral tends to "wind-up" if there are large errors that persist for a while. This can cause problems with controllers that include an **integral** term. Sometimes this tendency of the **integral** term to want to overshoot when it tries to correct the error is a big enough problem that the programmer must do something to the **integral** term so it won't cause problems. If **integral** wind-up is a problem two common solutions are (1) zero the **integral**, that is set the variable **integral** equal to zero, every time the **error** is zero or the **error** changes sign. (2) "Dampen" the integral by multiplying the accumulated **integral** by a factor less than one when a new **integral** is calculated. For example;

integral = (2/3)*integral + error

This reduces the previous integral value by 1/3 each time through the loop. If you think of the integral term as the controllers "memory" then this damping is forcing it to become forgetful of things that happened a "long" time ago.

Pseudo code for the PI controller

To add the integral term to the controller we need to add a new variable for **Ki** and one for the **integral** itself. And don't forget that we are multiplying our Ks by 100 to help with the integer math restrictions.

```
Kp = 1000              ! REMEMBER we are using Kp*100 so this is really 10 !
Ki = 100               ! REMEMBER we are using Ki*100 so this is really 1 !
offset = 45                  ! Initialize the variables
Tp = 50
integral = 0                 ! the place where we will store our integral
Loop forever
  LightValue = read light sensor   ! what is the current light reading?
  error = LightValue - offset      ! calculate the error by subtracting the offset
  integral = integral + error      ! our new integral term
  Turn = Kp*error + Ki*integral    ! the "P term" and the "I term"
```

```
Turn = Turn/100                    ! REMEMBER to undo the affect of the factor of 100 in Kp !
powerA = Tp + Turn                 ! the power level for the A motor
powerC = Tp - Turn                 ! the power level for the C motor
MOTOR A direction=forward power=powerA      ! actually issue the command in a MOTOR block
MOTOR C direction=forward power=powerC      ! actually issue the command in a MOTOR block
end loop forever                   ! done with this loop, go back to the beginning and do it again.
```

Adding “D” To The Controller: The Full PID Controller (“D”: what is going to happen next?)

Our controller now contains a proportional (P) term that tries to correct the current **error** and an **integral** (I) term that tries to correct past **errors** is there a way for the controller to look ahead in time and perhaps try to correct **error** that hasn’t even occurred yet?

Yes, and the solution is another concept from advanced mathematics called the **derivative**. Ahhh, there’s the “D” in PID. Like the **integral**, the **derivative** can represent some pretty serious mathematics. Fortunately for us, what we need for the PID is fairly simple.

We can look into the future by assuming that the next change in the **error** is the same as the last change in the **error**.

That means the next **error** is expected to be the current **error** plus the change in the **error** between the two preceding sensor samples. The change in the **error** between two consecutive points is called the **derivative**. The **derivative** is the same as the slope of a line.

That might sound a bit complex to calculate but it really isn’t too bad. A sample set of data will help illustrate how it works. Lets assume that the current **error** is 2 and the **error** before that was 5. What would we predict the next error to be? Well, the change in error is the **derivative** which is;

(the current **error**) - (the previous **error**)

which for our numbers is 2 - 5 = -3. The current **derivative** therefore is -3. To use the **derivative** to predict the next **error** we would use

(next error) = (the current **error**) + (the current **derivative**)

which for our numbers is 2 + (-3) = -1. So we predict the next **error** will be -1. In practice we don’t actually go all the way and predict the next **error**. Instead we just use the **derivative** directly in the controller equation.

The D term, like the I term, should actually include a time element, and the “official” D term is;

Kd(derivative)/(dT)

Just as with the **proportional** and **integral** terms we have to multiply by a constant. Since this is the constant that goes with the **derivative** it is called **Kd**. Notice also that for the derivative term we divide by **dT** whereas in the **integral** term we had multiplied by **dT**. Don’t worry too much about why that is since we are going to do the same kinds of tricks to get rid of the **dT** from the **derivative** term as we did for the **integral** term. The fraction **Kd/dT** is a constant if **dT** is the same for every loop. So we can replace **Kd/dT** with another **Kd**. Since this K, like the previous Ks, is unknown and has to be determined by trial and error it doesn’t matter if it is **Kd/dT** or just a new value for **Kd**.

We can now write the complete equation for a PID controller:

Turn = Kp*(error) + Ki*(integral) + Kd*(derivative)

It is pretty obvious that “predicting the future” would be a handy thing to be able to do but how exactly does it help? And how accurate is the prediction?

If the current **error** is worse than the previous **error** then the D term tries to correct the **error**. If he current **error** is better than the previous **error** then the D term tries to stop the controller from correcting the **error**. It is the second case that is particularly useful. If the **error** is getting close to zero then we are approaching the point where we want to stop correcting. Since the system probably takes a while to respond to changes in the motors’ power we want to start reducing the motor power before the **error** has actually gone to zero, otherwise we will overshoot. When put that way it might seem that the equation for the D term would have to be more complex than it is, but it isn’t. The only thing you have to worry about is doing the subtraction in the correct order. The correct order for this type of thing is “current” minus “previous”. So to calculate the **derivative** we take the current **error** and subtract the previous **error**.

Pseudo code for the PID controller

To add the derivative term to the controller we need to add a new variable for **Kd** and a variable to remember the last **error**. And

don't forget that we are multiplying our Ks by 100 to help with the integer math.

```
Kp = 1000          ! REMEMBER we are using Kp*100 so this is really 10 !
Ki = 100           ! REMEMBER we are using Ki*100 so this is really 1 !
Kd = 10000         ! REMEMBER we are using Kd*100 so this is really 100!
offset = 45        ! Initialize the variables
Tp = 50
integral = 0       ! the place where we will store our integral
lastError = 0      ! the place where we will store the last error value
derivative = 0     ! the place where we will store the derivative
Loop forever
  LightValue = read light sensor     ! what is the current light reading?
  error = LightValue - offset        ! calculate the error by subtracting the offset
  integral = integral + error        ! calculate the integral
  derivative = error - lastError     ! calculate the derivative
  Turn = Kp*error + Ki*integral + Kd*derivative     ! the "P term" the "I term" and the "D term"
 Turn = Turn/100     ! REMEMBER to undo the affect of the factor of 100 in Kp, Ki and Kd!
  powerA = Tp + Turn  ! the power level for the A motor
  powerC = Tp - Turn  ! the power level for the C motor
  MOTOR A direction=forward power=PowerA     ! actually issue the command in a MOTOR block
  MOTOR C direction=forward power=PowerC     ! same for the other motor but using the other power level
  lastError = error   ! save the current error so it can be the lastError next time around
end loop forever      ! done with loop, go back and do it again.
```

We now have the pseudo code for our complete PID controller for a line following robot. Now comes what is often the tricky part, "tuning" the PID. Tuning is the process of finding the best, or at least OK, values for **Kp**, **Ki** and **Kd**.

Tuning A PID Controller Without Complex Math (but we still have to do some math)

Very smart people have already figured out how to tune a PID controller. Since I'm not nearly as smart as they are, I'll use what they learned. It turns out that measurement of couple of parameters for the system allows you to calculate "pretty good" values for **Kp**, **Ki** and **Kd**. It doesn't matter much what the exact system is that is being controlled the tuning equations almost always work pretty well. There are several techniques to calculate the Ks, one of is called the "*Ziegler–Nichols Method*" which is what we will use. A google search will locate many web pages that describe this technique in all it's gory detail. The version that I'll use is almost straight from the Wiki page on PID Controllers (the same treatment is found in many other places). I'll just make one small change by including the loop time (**dT**) in the calculations shown in the table below.

To tune your PID controller you follow these steps:
1. Set the **Ki** and **Kd** values to zero, which turns those terms off and makes the controller act like a simple P controller.
2. Set the **Tp** term to a smallish one. For our motors 25 might be a good place to start.
3. Set the **Kp** term to a "reasonable" value. What is "reasonable"?
    1. I just take the maximum value we want to send to the motor's power control (100) and divide by the maximum useable error value. For our line following robot we've assumed the maximum error is 5 so our guess at **Kp** is 100/5=20. When the error is +5 the motor's power will swing by 100 units. When the error is zero the motor's power will sit at the **Tp** level.
    2. Or, just set **Kp** to 1 (or 100) and see what happens.
    3. If you have implemented that the K's are all entered as 100 times their actual value you have to take that into account here. 1 is entered as 100, 20 as 2000, 100 as 10000.
4. Run the robot and watch what it does. If it can't follow the line and wanders off then increase **Kp**. If it oscillates wildly then decrease **Kp**. Keep changing the **Kp** value until you find one that follows the line and gives noticeable oscillation but not really wild ones. We will call this **Kp** value "Kc" ("critical gain" in the PID literature).
5. Using the Kc value as **Kp**, run the robot along the line and try to determine how fast it is oscillating. This can be tricky but fortunately the measurement doesn't have to be all that accurate. The oscillation period (**Pc**) is how long it takes the robot to swing from one side of the line to the other then back to the side where it started. For typical LEGO robots **Pc** will probably be in the range of about 0.5 seconds to a second or two.
6. You also need to know how fast the robot cycles through it's control loop. I just set the loop to a fixed number of steps (like 10,000) and time how long the robot takes to finish (or have the robot do the timing and display the result.) The time per loop (**dT**) is the measured time divided by the number of loops. For a full PID controller, written in NXT-G, without any added buzzes or whistles, the **dT** will be in the range of 0.015 to 0.020 seconds per loop.
7. Use the table below to calculate a set of **Kp**, **Ki**, and **Kc** values. If you just want a P controller then use the line in the table marked P to calculate the "correct" **Kp** (**Ki**' and **Kd**' are both zero). If you want a PI controller then use the next line. The full PID controller is the bottom line.
8. If you have implemented that the K's are all entered as 100 times their actual value you don't have to take that into account in these calculations. That factor of 100 is already take into account in the **Kp** = Kc value you determined.
9. Run the robot and see how it behaves.
10. Tweak the **Kp**, **Ki** and **Kd** values to get the best performance you can. You can start with fairly big tweaks, say 30% then try smaller tweaks to get the optimal (or at least acceptable) performance.
11. Once you have a good set of K's try to boost the **Tp** value, which controls the robot's straight speed.
12. Re-tweak the K's or perhaps even go back to step 1 and repeat the entire process for the new Tp value.

13. Keep repeating until the robot's behavior is acceptable.

Ziegler–Nichols method giving K' values (loop times considered to be constant and equal to dT)				
Control Type	Kp	Ki'	Kd'	
P	0.50Kc	0	0	
PI	0.45Kc	1.2KpdT/ Pc	0	
PID	0.60Kc	2KpdT / Pc	KpPc / (8dT)	

The primes (apostrophes) on the **Ki**' and **Kd**' are just to remind you that they are calculated assume **dT** is constant and **dT** has been rolled into the K values.

I couldn't find the equations for the PD controller. If anyone knows what they are please send me an email.

Here are the values I measured for my test robot (the one in the video linked later on). Kc was 300 and when **Kp**=Kc the robot oscillated at about 0.8 seconds per oscillation so **Pc** is 0.8. I measured Pc by just counting out loud every time the robot swung fully in a particular direction. I then compared my perception of how fast I was counting with "1-potato -- 2-potato -- 3-potato ...". That's hardly "precision engineering" but it works well enough so we'll call it "practical engineering". The loop time, **dT**, is 0.014 seconds/loop determined by simply running the program for 10,000 loops and having the NXT display the run time. Using the table above for a PID controller we get;

Kp = (0.60)(Kc) = (0.60)(300) = 180
Ki = 2(**Kp**)(**dT**) / (**Pc**) = 2(180)(0.014) / (0.8) = 6.3 (which is rounded to 6)
Kd = (**Kp**)(**Pc**) / ((8)(**dT**)) = (180)(0.8) / ((8)(0.014)) = 1286

After further trial and error tuning the final values were 220, 7, and 500 for **Kp**, **Ki** and **Kd** respectively. Remember that all of my K's are entered as 100x their actual value so the actual values are 2.2, 0.07 and 5.

How changes in Kp, Ki, and Kd affect the robots behavior

The table and method described above is a good starting point for optimizing your PID. Sometimes it helps to have a better idea of what the result will be of increasing (or decreasing) one of the three Ks. The table below is available from many web sites. This particular version is from the Wiki on PID controllers.

Effects of increasing parameters				
Parameter	Rise time	Overshoot	Settling time	Error at equilibrium
Kp	Decrease	Increase	Small change	Decrease
Ki	Decrease	Increase	Increase	Eliminate
Kd	Indefinite (small decrease or increase)	Decrease	Decrease	None

The "Rise Time" is how fast the robot tries to fix an error. In our sample case it is how fast the robot tries to get back to the line edge after it has drifted off of it. The rise time is mostly controlled by **Kp**. A larger **Kp** will make the robot try to get back faster and decreases the rise time. If **Kp** is too large the robot will overshoot.

The "Overshoot" is how far past the line edge the robot tends to go as it is responding to an error. For example, if the overshoot is small then the robot doesn't swing to the right of the line as it is trying to fix being to the left of the line. If the overshoot is large then the robot swings well past the line edge as it tries to correct an error. Overshoot is largely controlled by the **Kd** term but is strongly affected by the **Ki** and **Kp** terms. Usually to correct for too much overshoot you will want to increase **Kd**. Remember our first very simple line follower, the one that could do nothing but turn right or left? That line follower has very bad overshoot. Indeed that is about all it does.

The "settling time" is how long the robot takes to settle back down when it encounters a large change. In our line following case a large change occurs when the robot encounters a turn. As the robot responds to the curve it will correct the **error** and then overshoot by some amount. It then needs to correct the overshoot and might overshoot back the other way. It then needs to correct the overshoot ... well, you get the idea. As the robot is responding to an **error** it will tend to oscillate around the desired position. The "settling time" is how long that oscillation takes to dampen out to zero. The settling time responds strongly to both the **Ki** and **Kd** terms. Bigger **Ki** gives longer settling times. Bigger **Kd** gives shorter settling time.

"Error at Equilibrium" is the error remaining as the system operates without being disturbed. For our line follower it would be the

offset from the line as the robot follows a long straight line. Often P and PD controllers will end up with this kind of error. It can be reduced by increasing **Kp** but that may make the robot oscillate. Including an I term and increasing **Ki** will often fix a P or PD controller that has a constant error at equilibrium. (This assumes you even care about a small remaining error as the robot follows the line. It just means it is offset to one side or the other by a small amount.)

How well does it work?

Here's a short video of a basic LEGO Mindstorms robot following the line on the test mat that comes with the set. The video quality isn't very good.

The light sensor is about 1/2" above the mat and offset to one side of the robot's center line. The **Tp** (target power) was set at 70%. The robot averages about 8 inches per second on this course. The robot is a left hand line follower and is following the inside edge of the oval. The inside edge is a bit harder to follow than the outside edge.

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

Overall the line follower appears to work pretty well. If you watch the video closely you'll see the robot "wag its tail" a bit as it comes off the corners. That's the PID oscillating a little. When the robot is running towards the camera you can see the red spot on the mat from the light sensor's LED. It looks to be tracking the line's edge pretty well.
The basic PID controller should work for many different control problems, and of course can be used as a P or PI controller instead of a PID. You would need to come up with a new definition of the error and the PID would have to be tuned for the particular task.
#