A PID Controller For LEGO® Mindstorms Robots (Part 1)

We present you the first tutorial suggested by one of our readers, we hope you find it of interest and you'll send us suggestions.

Text and pictures by J. Sluka

A PID Controller is a common technique used to control a wide variety of machinery including vehicles, robots and even rockets. The complete mathematical description of a PID Controller is fairly complex but a much simpler understanding is really all that is needed to use a PID effectively.

This document is a description of how to create a PID Controller for use with LEGO® Mindstorms Robots using the NXT-G programming language.

It will be easier if we have an actual task in mind so I will describe how to create a PID to do line following. Once created, the same PID can be used, with only minor modifications, with any other Mindstorms application such as getting a robot that can drive as straight as possible, or even for a robot that can balance with nothing but 2 wheels touching the ground like a Segway. A PID is really pretty straight forward and the typical description of a PID is easily understood by anyone that has had Calculus. This document is targeted towards First LEGO League kids in third through eighth grade. *Since there aren't many kids that have had Calculus I'll try to build the whole concept up from a very simple starting point without using any Calculus.*

So lets start with the basic layout of a robot that would be suitable for line following. At the right is a simplified drawing of a top view of the robot with all the details we need. The robot is a differential steer robot with two motors, each connected to one of the wheels A and C. The robot has a light sensor mounted at the front that points straight down so it sees nothing but the mat (floor, ground, table top, whatever the robot is on). The red circle represents the fairly small spot on the mat that the light sensor can actually "see". The rest of the robot is the large rectangle with an arrow, the arrow shows the normal direction of travel.

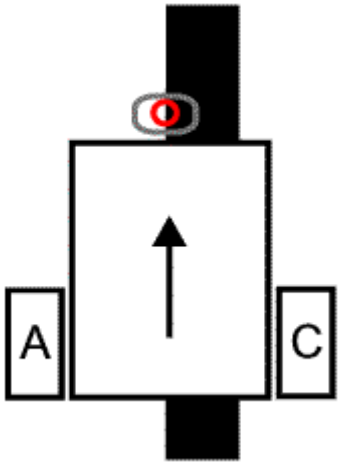


Figure 1

Our goal is to get the robot to follow the fat black line. Line following is a basic robotic behavior and is often one of the first things people learn. A mobile device that can follow a line displays all the characteristics of a true robot. It uses sensor to gather information about the world around it and changes it's behavior depending upon that information.

Line followers can be built with one light sensor, or two, or a dozen or however many you happen to have. In general, the more light sensors you have the better you can follow a line. Here we'll limit ourselves to a single Mindstorms light sensor. Even with a single sensor we should be able to build a robot that can track the line very precisely even if the line has curves in it. The thing you usually loose with a single sensor is the ability to follow the line while moving fast. Often, the more sensors you have the faster the robot can move while it follows the line.

The first trick we'll use, which is unrelated to a PID, is that we won't try to actually follow the line. Instead, we'll try to follow the edge of the line. Why? Because if we follow the line itself (the black) then when the robot drifts off the line and the sensor "sees white" we don't know which side of the line we are on. Are we left or right of the line? If we follow the line's edge then we can tell which way we are off the edge as the robot drifts off the line. If the light sensor "sees white" then we know it is left of the line's edge (and the line). If it "sees black" then we know it is to the right of the line's edge (and on the line). This is called a "*left hand line follower*" since it is following the line's left edge.

We need to know what values the light sensor returns when it "sees white" and when it "sees black". A typical uncalibrated sensor might give a "white" reading of 50 and a "black" reading of 40 (uncalibrated, on a 0 to 100 scale). It is convenient to draw the values on a simple number line to help visualize *how we convert light sensor values into changes in the robot's movement*. Below are our made up light values for white and black.

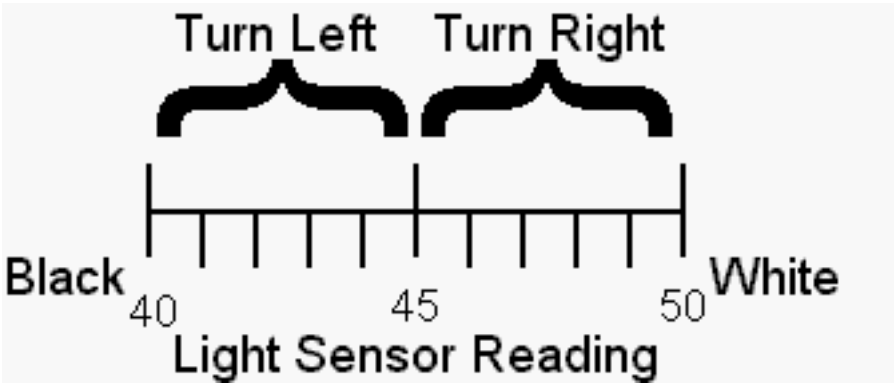


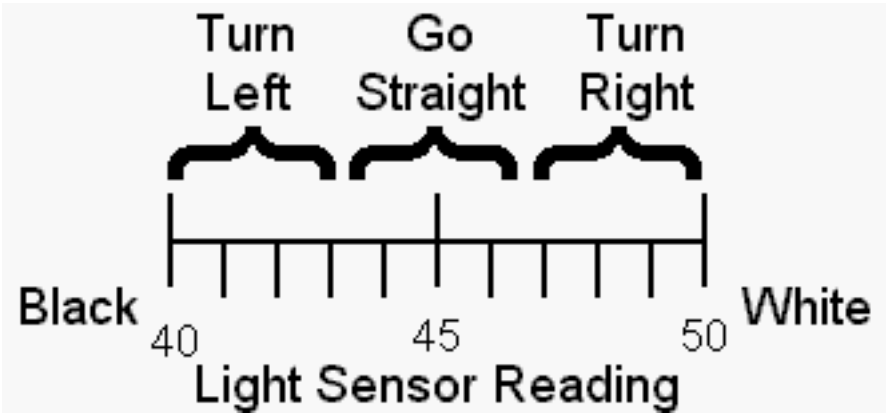
Figure 2

We'll just divide the range into two equal pieces and say that if the light level is less than 45 we want the robot to turn left. If it is greater than 45 we want to turn right. I won't go into how exactly the turns should be done. I'll just say that gentle turns work well for a fairly straight line. A line with lots of curves usually needs to be making sharper turns. For gentle turns you might use Power levels of 50% on the fast wheel and 20% on the slow wheel. For sharper turns on a curvy line you might need to use 30% power for the fast wheel and coast or brake the slow wheel. Whatever power levels you use the numbers will be the same for the two turns, you just switch which motor gets the big number and which get the smaller number (or a stop command).

This type of a line follower will follow a line but it isn't very pretty. It looks OK on a straight line with with the motors programmed for gentle turns. But if the line has any curves then you have tell the robot to use sharper turns to follow line. That makes the robot swing back and forth across the line. The robot only "knows" how to do two things; turn left and turn right. This approach can be made to work but it is not very fast or accurate and looks terrible.

In the above approach the robot never drives straight, even if it is perfectly aligned with line's edge and the line is straight. That doesn't seem very efficient does it?

Lets try to fix that. Instead of dividing our light value number line into two regions lets divide it into three.



So now if the light level is less than 43 we want the robot to turn left. If the light value is between 44 and 47 we want it to go straight (zoom zoom). If the light level is greater than 47 we want to turn right. This can be easily be implemented in Mindstorms NXT-G with a switch (yes/no) within a switch. You actually only have to do two tests not three.

This approach works better than the first one. At least now the robot is sometimes moving straight forward. As with the first approach you still have to decide what kinds of turns you need and that usually depends on the characteristics of the line you are following. The robot will probably still hunt back a forth a fair amount.

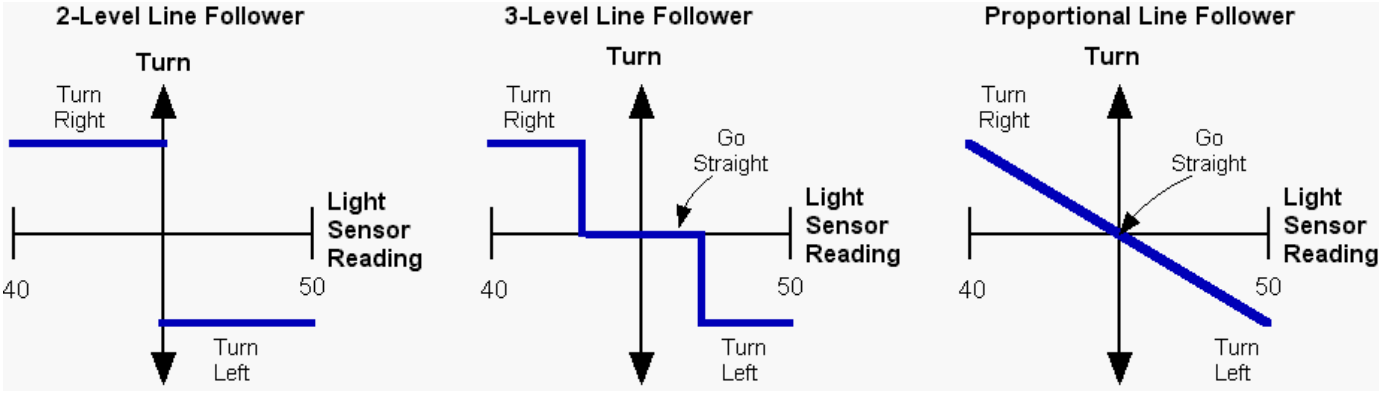
The astute reader will probably have thought "*well if three light ranges are better than two than what about adding even more?*" That is the beginning of a PID.

The "P" in "PID": Proportion(al) is the key

So what will happen if we add more divisions to our light scale line? Well the first thing we have to deal with is what does "turn" mean with more than three light ranges? In our first approach the robot could do just two things, turn left or right. The turns were always the same just in opposite directions. In the second approach we added the "go straight" to the two turns. If we have more than three ranges then we need more "kinds" of turns.

To help understand "more kinds of turns" we will redo or number line a bit and convert it into a graph. Our X-axis (horizontal) will be our light values just like on the number lines. The Y-axis (vertical) well be our "turn" axis.

Figure 3



On the left is our original two level setup expressed on a graph. The robot can only do two things (shown by the blue lines), turn right or left and the turns are always the same except for their direction. In the center is the three level follower. The added center range is where the robot drives straight (Turn=0). The turns are the same as before. On the right is a **Proportional** line follower. In a proportional line follower the turn varies smoothly between two limits. If the light sensor reading says we are close to the line then we do a small turn. If we are far from the line then we do a big turn. Proportional is an important concept. Proportional means there is a linear relationship between two variables. To put it even simpler, proportional means a graph of the variables against each other produces a straight line (as in the right hand graph above).

As you may know, the equation of a straight line is:

$y = mx + b$

Where y is the distance up (or down) the Y-axis, x the distance on the X-axis, m is the slope of the line and b is the Y intercept, the point where the line crosses the Y-axis when x is zero. The slope of the line is defined as the change in the y value divided by the change in the x value using any pair of points on the line.

If you don't know much about lines (or have forgotten what you once new) I'll expand a bit and make some simplifications to our graph and equation. First, we will shift the center of our light number line (the X-axis) to zero. That's easy to do. For our 40 and 50 light value range we just subtract 45 (that's the average of 40 and 50, (40+50)/2) from all of our light readings. We will call that result the **error**. So, if the light value is 47 we subtract 45 and get an **error**=2. The error tells us how far off the line's edge we are. If the light sensor is exactly on the line's edge our **error** is zero since the light value is 45 and we subtract 45 from all of our readings. If the sensor is all the way out into the white our **error** is +5. All the way into the black the **error** is -5.

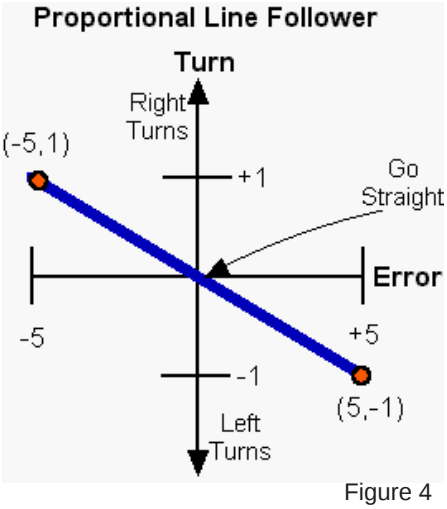


Figure 4

In the above graph I have shifted the axis by converting it to an error scale. Since the line now crosses the Y-axis at zero that means b is is zero and the equation for the line is a bit simpler;

y = mx

or using our labels

Turn = m***error**

We haven't yet defined what the turn axis means so for now we will just say the turns range from -1 (hard turn to the left) to +1 (hard turn to the right) and a zero turn means we are going straight. The slope of the line in the graph above can be calculated using the two points marked in red (any two points on the line will work);

slope = m = (change in y)/(change in x) = (1- (-1)) / (-5 - 5) = 2/10 = 0.2

The slope is a **proportionality constant** and is the factor that you have to multiply the **error** (x value) by to convert it into a **Turn** (y value). That's an important thing to remember.

The "slope" has a couple names that all mean the same thing, at least in this context. In the PID literature slopes (proportionality constants, m in the equation of a line) are called "**K**" (from misspelling of the word "constant"?). Various **K**s show up all over the PID literature and are a very important. You can think of a **K** (or m or slope or proportionality constant) as a conversion factor. You use **K** to convert a number that means one thing (light values or error in our case) into something else like a turn. That's all that a **K** does. Very simple and very powerful.

So using these new names for our variables the equation of the line is;

Turn = K*(error)

In words that's "take the **error** and multiply it by the proportionality constant **K** to get the needed turn. The value **Turn** is the output of our P controller and is called the "**P term**" since this is only a proportional controller.

You may have noticed that in the last graph the line does not extend outside the **error** range of -5 to +5. Outside the range of -5 to +5 we can't tell how far the sensor is from the line. All "white" looks the same once the sensor can't see any black at all. Remember that this range is arbitrary, your range will be determined by you light sensor setup, the colors of the mat etc. Once the light sensor gets too far from the line edge it starts to give a constant reading, that means the light sensor reading is no longer proportional to the **error**. We can only judge how far the sensor is from the line's edge when the sensor is actually pretty close to it. Over that narrow range the light sensor reading is proportional to the distance. So our sensor setup has a limited range over which it gives proportional information. Outside that range it tells us the correct direction but the magnitude (size) is wrong. The light sensor reading, or the **error**, is smaller than it should be and doesn't give as good an idea of what the turn should be to fix the **error**.

In the PID literature the range over which the sensor gives a proportional response is called the **"proportional range"** (go figure :D). The proportional range is another very important concept in PIDs. In our line follower the proportional range for the light sensor is 40 to 50, for the **error** it is -5 to +5. Our motors also have a proportional range, from -100 (full power backwards) to +100 (full power forwards). I'll just say a couple things about it the importance of the proportional range:

(1) You want the proportional range to be as wide as possible. Our light sensor's proportional range is pretty small, that is, the sensor has to be pretty close to the line edge to get proportional information. Exactly how wide the range is depends mostly on how high the sensors is above the mat. If the sensor is very close to the mat, say 1/16 inch, then the sensor is seeing a very small circle on the mat. A small side to side movement of the light sensor will swing the error from -5 to +5, that's all the way through our proportional range. You might say the sensor has "tunnel vision" and it can only see a very small part of the mat. The sensor has to be very close to the line edge to get a reading that isn't either "white" or "black". If the sensor is moved higher off the mat then it sees a larger circle on the mat. At a height of about 1/2 inch the light sensor appears to be looking at a circle on the mat that is about 1/2 inch across. With the sensor up this high the proportional range is much wider, since the light sensor only needs to stay within +/- 1/2 inch of the line edge to maintain a proportional output. Unfortunately, there are two drawbacks to a high light sensor. First, a high light sensor "sees", and responds to, the room lights much more than a low sensor. A high sensor also has less difference between black and white than a low sensor. At a sufficiently large distance black and white will give the same reading.

(2) Outside the proportional range the controller will move things in the correct direction but it will tend to under correct. The controller's proportional response is limited by the proportional range.

From P to actual motor power levels

How can we implement the turns? What should the actual motor power levels be? One way to do the turns is to define a "**T**arget **p**ower level", which I'll call "**Tp**". **Tp** is the power level of both motors when the robot is supposed to go straight ahead, which it does when the **error**=0. When the **error** is not zero we use the equation **Turn = K*(error)** to calculate how to change the power levels for the two motors. One motor will get a power level of **Tp+Turn**, the other motor will get a power level of **Tp-Turn**. Note that since our **error** is -5 to +5 that means **Turn** can be either positive or negative which corresponds to turns in opposite

directions. It turns out that that is exactly what we want since it will automatically set the correct motor as the fast motor and the other one as the slow motor. One motor (we'll assume it is the motor on the left of the robot plugged into port A) will always get the **Tp+Turn** value as it's power level. The other motor (right side of robot, port C) will always get **Tp-Turn** as it's power level. If error is positive then **Turn** is positive and **Tp+Turn** is greater than **Tp** and the left motor speeds up while the right motor slows down . If the error changes sign and becomes negative (meaning we have crossed over the line's edge and are "seeing black") then **Tp+Turn** is now less than **Tp** and the left motor slows down and the right motor speeds up since **Tp-Turn** is greater than **Tp**. (Remember that the negative of a negative is a positive). Simple eh? Hopefully it'll be a bit clearer as we go on.

Pseudo Code for a P Controller

First we need to measure the values the light sensor returns for white and black. From those two number we can calculate the **offset**, that is, how much to subtract from a raw light reading to convert it to an error value. The **offset** is just the average of the white and black readings. For simplicity I'll assume that the **offset** has already been measured and stored in a variable called **offset**. (A nice upgrade would be to have the robot measure the white and black levels and calculate the **offset**.)

We will also need a storage location for the **K** constant, we'll call that **Kp** (the **K**onstant for the proportional controller). And, an initial guess as to what **Kp** should be. There are a lot of ways to get that first **Kp** value. You can guess and then refine it by trial and error. Or, you can try to estimate a value based on the characteristics of the sensor and robot. We'll do the latter. We will use a **Tp** (target power) of 50, when the error is zero both motors will run at power level 50. The error ranges from -5 to +5. We'll guess that we want the power to go from 50 to 0 when the error goes from 0 to -5. That means the **Kp** (the slope remember, the change in y divided by the change in x) is;

Kp = (0 - 50)/(-5 - 0) = 10.

We will use the **Kp**=10 value to convert an **error** value into a **turn** value. In words our conversion is "for every 1 unit change in the **error** we will increase the power of one motor by 10". The other motor's power gets decreased by 10.

So, in pseudo code ("pseudo code" means this isn't actual NXT-G, or any other type of program code, instead it is just a detailed listing of what we want the program to do):

```
Kp = 10                          ! Initialize our three variables
offset = 45
Tp = 50
Loop forever
  LightValue = read light sensor     ! what is the current light reading?
  error = LightValue - offset        ! calculate the error by subtracting the offset
  Turn = Kp * error                  ! the "P term", how much we want to change the motors' power
  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 ! issue the command with the new power level in a MOTOR block
  MOTOR C direction=forward power=powerC ! same for the other motor but using the other power level
end loop forever                     ! done with this loop, go back to the beginning and do it again
```

That's it, well almost. There is a subtle problem that should be corrected. But give it a try anyway. If your robot appears to avoid the line edge, instead of trying to find it, the most likely cause is that you have swapped the turn directions. Change Kp to -10 and see what happens. If that fixes the turn directions then change Kp back to +10 and change the signs in the two power lines to;

```
powerA = Tp - Turn
powerC = Tp + Turn
```

There are two "tunable parameters" and one constant in this P controller. The constant is the **offset** value (the average of white and black light sensor readings). You'll need to write a short program to measure the light levels on your mat with your robot. You need a "black" and a "white" value. Calculate the average and put it into the P controller program in the **offset** variable. Almost all line followers require that you (or code written by you and executed by the robot) do this step.

The **Kp** value and the target power **Tp** are the tunable parameters. A tunable parameter has to be determined by trial and error. **Kp** controls how fast the controller will try to get back to the line edge when it has drifted away from it. **Tp** controls how fast the robot is moving along the line.

If the line is pretty straight you can use a large **Tp** to get the robot running at high speed and a small **Kd** so the turns (corrections) are gentle.

If the line has curves, especially sharp ones, there will be a maximum **Tp** value that will work. If **Tp** is bigger than that maximum it won't matter what **Kp** is, the robot will loose the line when it encounters a curve because it is moving too fast. If **Tp** is really small then almost any **Kp** value will work since the robot will be moving very slowly. The goal is to get the robot moving as fast as possible while still being able to follow the line of interest.

We had guesstimated a starting value for **Kp** of 10. For **Tp** you might start at even lower than suggested above, perhaps 15 (the robot will be moving pretty slow). Try it and see how it works. If you loose the line because the robot seems to turn sluggishly then increase **Kp** by a couple and try again. If you loose the line because the robot seems hyperactive in hunting back and forth for the

line then decrease **Kp**. If the robot seems to follow the line pretty well then increase **Tp** and see if you can follow the line at the faster speed. For each new **Tp** you will need to determine a new **Kp**, though **Kp** usually won't change too much.

Following a straight line is usually pretty easy. Following a line with gentle curves is a bit harder. Following a line with sharp curves is the hardest. If the robot is moving slow enough then almost any line can be followed, even with a very basic controller. We want to get good line following, good speed and the ability to handle gentle corners. (Lines with sharp corners usually take more specialized line followers.)

It is likely that the best P controller will be different for each kind of line (line width, sharpness of curves etc.) and for different robots. In other words, a P controller (or a PID controller for that matter) is tuned for a particular kind of line and robot and will not necessarily work well for other lines or robots. The code will work for many robots (and many tasks) but the parameters, **Kp**, **Tp** and **offset**, have to be tuned for each robot and each application.

Doing math on a computer that doesn't know what a decimal point is causes some problems

NOTE: This work was done using NXT-G version 1.1 which only supports integers. NXT-G version 2 supports floating point numbers so the following may not be needed if you have version 2 or later.

In the process of tuning the P controller you will be tweaking the **Kp** value up and down. The expected range of values that **Kp** might be depends on exactly what the P controller is doing. How big is the input range and how big is the output range? For our line follower P controller the input range is about 5 light units, and the output range is 100 power units, so it seems likely that **Kp** will be in the vicinity of 100/5=20. In some cases the expected **Kp** won't be that big. What happens if the expected **Kp** is one? Since variables in NXT-G are limited to integers, when you try to tune the **Kp** value all you can try is ...-2, -1, 0, 1, 2, 3, ... You can't enter 1.3 so you can't try **Kp**=1.3. You can't use any number with a decimal point! But there will probably be a large difference in the robot behavior when you change the **Kp** by the smallest possible change of 1 to 2. With **Kp**=2 the robot tries twice as hard to correct the error compared to **Kp**=1. The motor power level changes twice as much for the same change in the light levels. We really would like to have finer control of **Kp**.

It is pretty easy to fix this problem. All we will do is multiply the **Kp** by a power of ten to increase the usable range within the integer restriction. If it is expected that **Kp** might be near 1 then a value of 100 as the multiplier would be a good bet. Indeed, it is probably best to just go ahead and always use 100***Kp** as the number you actually enter into the program. Once **Kp** has been multiplied by 100 we can now enter what would have been 1.3 as 130. 130 has no decimal point so NXT-G is happy with the number.

But doesn't that trash the calculation? Yes it does but it is easy to fix. Once we have calculated the P term we will divide by 100 to remove our multiplier. Remember our equation that defines the P controller from earlier;

Turn = Kp*(error)

We will multiply **Kp** by 100, which means our calculated **Turn** is 100 times bigger than it should be. Before we use **Turn** we must divide it by 100 to fix that.

So, our new and improved pseudo code for a line following P controller is:

```
Kp = 1000                          ! REMEMBER we are using Kp*100 so this is really 10 !
offset = 45                        ! Initialize the other two variables
Tp = 50
Loop forever
  LightValue = read light sensor       ! what is the current light reading?
  error = LightValue - offset          ! calculate the error by subtracting the offset
  Turn = Kp * error                    ! the "P term", how much we want to change the motors' power
 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 ! same for the other motor but using the other power level
end loop forever                       ! done with loop, go back and do it again.
```

Wait, what was the "Subtle Problem" you mentioned with the first version of the P controller?

There are always subtle problems. Sometime they matter and sometimes they don't. ;)
In this case, one problem is that when we calculate the motor power level (e.g., powerC=Tp-Turn) it is possible to get a negative number for the power. We want a negative number to mean that the motor should reverse direction. But the data port on a NXT-G MOTOR block doesn't understand that. The power level is always a number between 0 and +100. The motor's direction is controlled by a different input port. To get the motor to react correctly when the power is negative you'll need to handle it in the program. Here is one way to do that;

```
If powerA > 0 then                     ! positive motor power is no problem
  MOTOR A direction=forward power=powerA
else
```

```
powerA = powerA * (-1)                ! negative motor power needs to be made into
  MOTOR A direction=reverse power=powerA  ! a positive number and the motor direction
end If                                ! needs to be reversed on the control panel
```

The MOTOR block receives the power (powerA for the A motor) via a data wire. The direction is set with the check boxes in the motor's parameter window.

You will need a similar chunk of code for the C motor. Now when the calculated power goes negative the motors will be properly controlled. One thing this does is allow the P controller to go all the way to a "zero turning radius turn" and the robot can spin in place if needed. Of course, that may not actually help.

There are a couple other things that might be subtle problems. What happens when you send a power level that is greater than 100 to the motor? It turns out that the motor just treats the number as 100. That is good for the program but not the best thing to have happen in a P (or PID) controller. You would really prefer that the controller never tries to ask the motors to do something they can't. If the requested power isn't too far above 100 (or below -100) then you are probably OK. If the requested power is a lot bigger than 100 (or a lot less than -100) then it often means the controller is spiraling out of control. So, make sure you have a fire extinguisher handy!

P Controller Summary

Hopefully you've picked up enough to understand a P (proportional) controller. It is pretty simple. Use a sensor to measure something that you are trying to control. Convert that measurement to an **error**. For the line follower we did that by subtracting the average of black and white light values. Multiply the **error** by a scaling factor called **Kp**. The result is a correction for the system. In our line follower example the correction is applied as an increase/decrease in the power level of the motors. The scaling factor **Kp** is determined using a bit of educated guessing and then fine tuned by trial and error.

P controllers can handle a surprising wide range of control problems, not just following a line with a LEGO robot. In general, P controllers work very well when a few conditions are met.

1. The sensor needs to have wide dynamic range (which unfortunately is not true for our line following robot).

2. The thing being controlled (motors in our case) should also have a wide dynamic range, that is they should have a wide range of "power" levels with individual "power" levels that are close together (the NXT motors are pretty good in this respect).

3. Both the sensor and the thing being controlled must respond quickly. "Quick" in this case is "much faster than anything else that is happening in the system". Often when you are controlling motors it isn't possible to get "quick" response since motors take time to react to a change in power. It can take a few tenths of a second for LEGO motors to react to a change in power levels.

That means the robot's actions are lagging behind the P controller's commands. That makes accurate control difficult with a P controller.

So where's the code?

I could give it to you but then I'd have to kill you.

Since this document is targeted at older FLL kids, I really don't want to give'm the code. They should be able to write their own.

The pseudo code has pretty much everything you need for the PID itself. You may have to add some setup code and perhaps a suitable way of stopping the line follower loop.

As a little bit of help here's a MyBlock that takes two inputs, the target power Tp and the Turn value, and controls the two motors.

This MyBlock also properly deals with negative power levels. It even beeps when a motor reverses directions, which is handy for tuning. A properly tuned line following PID should rarely have to reverse motor directions.

PID_LF_MotorControl.rbt is the RBT file for NXT-G v1.1.
http://www.inpharmix.com/jps/_images/PID_LF_MotorControl.rbt

A screen shot of the program is at PID_LF_MotorControld.png
http://www.inpharmix.com/jps/_images/PID_LF_MotorControld.png

If you would **really** like to get my PID NXT-G code send me an email to Lego at InPharmix dot com
#