An introduction and tutorial for PID controllers, by George Gillard

Quick links in this guide:

Introduction:

So, what is PID? It is basically a method used in programming and if tuned properly, can be incredibly effective and accurate. PID stands for **P**roportional **I**ntegral **D**erivative, 3 separate parts joined together, though sometimes you don't need all three. For example, you could instead have just P control, PI control or PD control. Throughout this document I have included examples of pseudocode – an informal language, not real code, but close enough.

A bit of history...

PID controllers date to 1890s governor design. PID controllers were subsequently developed in automatic ship steering. One of the earliest examples of a PID-type controller was developed by Elmer Sperry in 1911, while the first published theoretical analysis of a PID controller was by Russian American engineer Nicolas Minorsky. Minorsky was designing automatic steering systems for the US Navy, and based his analysis on observations of a helmsman, observing that the helmsman controlled the ship not only based on the current error (distance/value remaining), but also on past error and current rate of change; this was then made mathematical by Minorsky. His goal was stability, not general control, which significantly simplified the problem. While proportional control provides stability against small disturbances, it was insufficient for dealing with a steady disturbance, notably a stiff gale (due to droop), which required adding the integral term. Finally, the derivative term was added to improve control.

– *Adapted from wikipedia.*

PID controllers are used widely in industry. In fact, 95% of all closed loop systems in industrial processes are run off PID control.

So, now you know a brief background, let's get on with explaining it all.

Last edited: 10 May, 2012

P – Proportional.

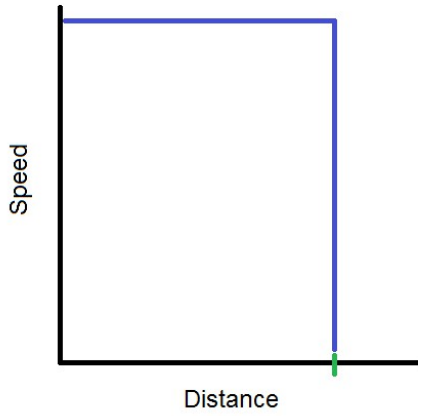
Imagine a robot that travels at full speed, for let's say, a value reading 1000 on the sensor used. Now, because of it's speed and inertia, it will probably overshoot a little and travel further than 1000 on the sensor. This can be a real nuisance when writing a program, as you want as much accuracy as possible.

This issue can be shown on a graph (**The green mark on the x axis represents the distance wanted to travel**):

In a perfect world, you would be able to tell the robot to stop and it would stop exactly where it was... (using code such as the pseudocode example below)

---Pseudocode---

```
motor = full speed;
while (travelled less than wanted)
{
}
motor = stop;
```



However, we aren't in a perfect world, and we have overshooting problems if we tell it to stop suddenly. By telling it to stop suddenly, here is what the result might look like:

… It will overshoot.

Now this overshoot wouldn't be a problem if the distance it overshot was always the same. However, there are lot's of variables that can change the distance it overshoots. Here are a few of the variables:

- Battery voltage. If the battery is low, the motors won't run as fast and so the robot will have less inertia. In this case, the robot will overshoot less.
- If the robot hits something, the overshoot will become less.
- If something pushes the robot in the direction it wants to travel, the overshoot will become greater.

So as you can see, overshoot is not good. **So the P controller controls the speed smoothly, allowing it to slow down as it approaches it's target**, to shrink the overshoot. That's why it is called a proportional controller – the output speed is *proportional* to the value remaining to be changed, which we call an **error**.

Here's how its done...

You have a nice variable, called an **error**. This is going to be the value according to the sensor, that is remaining to be changed, like I was telling you. For example, the error could be the distance remaining to be travelled, the height remaining to be lifted to, the temperature remaining to be heated up to, etc.

To calculate the error, we simply subtract the reading our sensor is giving us, from the value we want the sensor to tell us once it has finished.

error = (target value) – (sensor reading)

So, by giving some example values for the distance you want it to travel, and the distance it already has travelled, you will see the error gets smaller and smaller as it approaches its target.

Here are a few example values:

Target value	Current sensor reading	Error
1000	200	800
1000	400	600
1000	600	400
1000	800	200
1000	1000	0

We could use this to then control the applications speed, if you wanted to use a simple P controller, without the I and D terms. For this, we could write (shown on next page):

---Pseudocode---

```
error = (target value) – (sensor reading);
speed = error;
```

Except, the values for the error might not be quite as we would like it to be. The values may be way too high, so it overshoots the target a lot. If it overshoots, it will then try and correct the overshoot (the error would go into negative numbers), so on a debugger or a window where you can see the values and what is happening, you would see the error oscillating, overshooting then over-correcting.

Alternatively, the values for the error could be far too small, but you generally don't run into this problem.

For either problem, you can change the error so it keeps its proportional factor, but you multiply (or divide) the error by another number, a constant. You can call this whatever you like, but it is generally known as “Kp” - the constant term for the proportional component.

---Pseudocode---

```
Kp = 0.5;

error = (target value) – (sensor reading);
speed = Kp * error;
```

The pseudocode example above shows a simple P controller. I have set Kp to 0.5, but that is just a number from the top of my head and not necessarily the correct value for Kp. The value for Kp will vary depending on what the output values should be, and what input values are given. We will get round to finding the correct value for Kp later on in this tutorial.

Now, of course, the error needs to be recalculated as the values change, so we need to put it all in a loop.

---Pseudocode---

```
Kp = 0.5;

while (condition)
{
error = (target value) – (sensor reading);
speed = Kp * error;
}
```

I - Integral

So the proportional part of the code has got it so that the error remaining is pretty small. Too small for the proportional section to make much of a difference. This is where the integral comes in. The integral is the *running sum of previous errors*. So when your error is very small, the integral comes into action, but how does it actually work?

The integral wants to get it so that it travels fast enough to shrink the error, but not too fast, because then it might run the risk of overshooting. The way it decides how fast to go is that it will gently accelerate. The integral can be calculated like this:

*integral = integral + error*dT;*

The above is setting it so that the *new* value for "integral" (left of the equals sign) equals the *previous* value for "integral", plus (the error * delta time). Ignore the delta time part for now, I will come to that later on.

The way that the integral increases can be shown in a table (using an error of 2 as an example):

Cycle #	***Previous value for integral***	***Error***	***New value for integral***
0	0	2	**2**
1	2	2	**4**
2	4	2	**6**
3	6	2	**8**
4	8	2	**10**

The numbers in bold (the new value for the integral) are increasing.

So how do we add this to the existing code? Well, we literally *add* it. So the speed is now:

```
speed = (Kp * error) + integral;
```

Providing your programming language uses the BEDMAS (A.K.A. BODMAS) rule, you shouldn't need the brackets around the (Kp * error). However, to make it clearer to read by anyone (including yourself), it is helpful to group the Kp with the error, since they belong with each other. It can be shown clearer by removing the spaces between Kp_*_error.

```
speed = Kp*error + integral;
```

And now the integral code

---Pseudocode---

```
Kp = 0.5;

while (condition)
{
error = (target value) – (sensor reading);
integral = integral + error;
speed = Kp*error + integral;
}
```

Just like with the proportional code, we need to add a constant term to the integral. I will use 0.2 as an example value for Ki, though just like with the Kp, it is just a number that came from the top of my head.

---Pseudocode---

```
Kp = 0.5;
Ki = 0.2;

while (condition)
{
error = (target value) – (sensor reading);
integral = integral + error;
speed = Kp*error + Ki*integral;
}
```

I told you before to ignore the bit about delta time, but I'll explain all about that now.

Integral = integral + error*dT;

Delta time is needed when the loop may not always be taking the same time to complete each cycle, but when the cycles all take the same amount of time to complete as each other, the dT can be merged into the Ki. If the cycles don't take the same amount of time to complete, simply get the code to time itself each cycle and then you can use that time as your delta time.

For this tutorial, we will assume that the time between cycles is always the same, so dT will be merged into Ki.

Problems with the integral...

Problem number 1:

When your error finally reaches 0, your integral will probably still be at a value which keeps the speed high enough to keep the error changing. The equation will only reach 0 by itself if it passes *past* an error of 0, so the negative error can subtract into the existing integral. So, if the speed is still high enough to keep the error changing, we have a problem, right? There is a very simple solution to this problem, and that is to reset the integral if the error reaches 0. This can be done as follows. The fix in the code is in bold text.

---Pseudocode---

```
Kp = 0.5;
Ki = 0.2;

while (condition)
{
error = (target value) – (sensor reading);
integral = integral + error;

if (error is 0)
{
integral = 0;
}

speed = Kp*error + Ki*integral;
}
```

Problem number 2:

It is known as integral wind-up. It can start by if you have a large error to travel, the integral will start to build up once the loop starts to run. So, by the time the integral needs to be used, it is already at a value far higher than is usable. There are simple solutions to this issue to, 3 of which I will address.

Solution #1 – Limit the value that the integral can reach. If it is reaching too high, why not just put a limit on it? A limit could be written as follows:

```
if (integral is greater than or equal to the maximum value)
{
integral = maximum value;
}
```

But, if the integral is too big but in it's negative form (I. E. making the speed reverse too fast), you would need to rewrite the same as above but for the negative version of the integral.

Solution #2 – Limit the range in which the integral is allowed to build up in. So if the error is too big for the integral to be useful in, we could just disable the integral for that area.

```
if ( error is greater than useful for the integral )
{
disable the integral (set the integral to 0);
}
```

But again, just like for in solution #1, you would need to rewrite the same but for the negative values for the integral. Or, if your programming language supports the use of an absolute tool, you could use that to make the code shorter, and perhaps, more simple too.

For those not familiar with the absolute tool... The absolute tool (generally written as “abs” in code) returns a given value as a positive integer. For example, of you got the absolute of (-)8, the value would be (+)8. Also, if you got the absolute of (+)4, the value would still be (+)4.

How you could implement the absolute tool into the code:

if (abs(error) is greater than useful for the integral)
{
disable the integral (set the integral to 0);
}

Solution #3 – Restrain the amount of time the integral is allowed to build up in. This would a little more complicated to program, but still possible.

For this tutorial, we'll just use solution #2, as it is the shortest (for it's simplicity) to write. The range for which it is suitable to calculate the integral within will be +/- 40, but that is just a random number. The full version of the code (if left as is would be known as “PI control” [proportional integral control]) would be something like this:

---Pseudocode---

```
Kp = 0.5;
Ki = 0.2;

while (condition)
{
error = (target value) – (sensor reading);
integral = integral + error;

if (error = 0)
{
integral = 0;
}

if ( abs(error) > 40)
{
integral = 0;
}

speed = Kp*error + Ki*integral;
}
```

D - Derivative

The final bit of the PID code – the derivative! The job of the derivative is to predict the future value for the error, and then make the speed act accordingly. For example, if it thinks it will overshoot, it will slow it down.

To be able to predict the next error, we need to know the previous error, and then find the difference between the two.

*derivative = (**(current error) – (previous error)**) / dT*

This will find the change between the current error and the previous error, and then we could use that to find the next error by adding it to the current error. Just like with the integral, the derivative is effected by dT, but providing that the time it takes to complete one cycle of the loop is always the same, the dT can be merged into Kd.

Here is a table showing examples of what the future error would likely be, calculated by the derivative:

Current error	Previous error	Next error (error + derivative)
50	55	45
20	30	10
2	3	1
5	15	-5

In our code, the derivative is calculated using the equation above, and then added to the speed (where is is also multiplied by Kd for scaling purposes).

We will need to create a new integer, named the previous error (or whatever you choose, as long as it represents the previous error's value), and we will get it to update itself *after* we have calculated the derivative. We can get it to update itself very simply, by telling it to set it's value to be the same as that of the error's.

Pseudocode for the full PID controller:

---Pseudocode---

```
Kp = 0.5;
Ki = 0.2;
Kd = 0.1;

while (condition)
{
error = (target value) – (sensor reading);
integral = integral + error;

if (error = 0)
{
integral = 0;
}

if ( abs(error) > 40)
{
integral = 0;
}

derivative = error – previous_error;

previous_error = error;

speed = Kp*error + Ki*integral + Kd*derivative;
}
```

What you see above is complete PID code. It may look a bit strange with the colours, but I have used them to help clarify the separate parts of the control. Blue represents proportional, red for the integral and green for the derivative.

Tuning the constant terms

This is the most time consuming and labour intensive. There are many different methods for tuning the Kp, Ki and Kd, I will try to explain a couple of them as best I can. Ways to tune the PID constants can be done by a computer program, by maths calculations or by manual tuning. I highly recommend at all times to view all the values for the error, speed etc. so you can see how close it is getting to the target point/how much is remaining to be changed. Use a debugger or a similar monitoring tool to check the results.

First of all, it is important to know the rules of tuning a PID controller. What effects are changed when each constant is increased is shown below in the table. The constants terms are in the column on the left, and the effects they have are listed along the top row. The effects are:

Rise time – the time it takes to get from the beginning point to the target point

Overshoot – the amount that is changed too much; the value further than the error

Settling time – the time it takes to settle back down when encountering a change

Steady-state error – the error at the equilibrium

Stability – the “smoothness” of the speed

What happens when each of the constants is increased:

Constant:	Rise time:	Overshoot:	Settling time:	Steady-state error:	Stability:
Kp	decrease	increase	Small change	decrease	degrade
Ki	decrease	increase	increase	decrease	degrade
Kd	minor change	decrease	decrease	No effect	Improve (if small enough)

Manual Tuning:

Manual tuning is done completely by you – there is no maths involved, but it can also be a bit inefficient at times. I personally use a manual tuning method though, as I can increase each term gently and know when something is going to be too high, whereas with maths methods such as the Ziegler-Nichols method, you never know quite how things will turn out until you try it. After all, in theory, practice and theory are the same, but in practise, they aren't.

The way I tune my constants is as follows:

1. Set Kp, Ki, and Kd to 0. This will disable them for now.
2. Increase Kp until the error is fairly small, but it still gets from the beginning to nearly the end quickly enough.
3. Increase Kd until any overshoot you may have is fairly minimal. But be careful with Kd – too much will make it overshoot.
4. Increase Ki until any error that is still existing is eliminated. Start with a *really* small number for Ki, don't be surprised if it is as small as 0.0001 or even smaller.
5. Using the rules of tuning the constants (in the table on the previous page), you can change around the constants a little bit to get it working to the best performance.

Knowing how big/small each constant term should be gets easier with practise, you will get a rough idea for it after practising a few times.

Tuning with maths (using the Ziegler-Nichols method):

In this method, the values for the constant terms are mainly found out by maths, but you will almost definitely need to tweak them a bit afterwards, to get it all working perfectly. The first bit of this method is done manually, but after you have done a few tests it is on with the maths.

1. Set Kp, Ki and Kd to 0. This will disable them for now.
2. Start increasing Kp until there is some oscillation in the error. Not too much oscillation, just a noticeable amount. Save this value for Kp as “Ku” - the value for the ultimate or critical gain.
3. You now need to measure the period of the oscillation. Save the value for the period as “Pu” - the period at the ultimate or critical gain.
4. Now you have done the most of the manual part of this method, so on with the maths of finding Kp, Ki and Kd! You will see down the left hand side column there are 3 “controller types”. P is a simple proportional controller, PI is proportional with integral, PID is the full proportional integral derivative code. There's no PD (proportional with derivative) in the table, but that doesn't say it doesn't exist. Here is the table for finding Kp, Ki and Kd:

Ziegler-Nichols Method for finding Kp, Ki and Kd:

Controller type:	**Kp:**	**Ki:**	**Kd:**
P	0.50Ku	N/A	N/A
PI	0.45Ku	1.2Kp/Pu	N/A
PID	0.60Ku	2Kp/Pu	KpPu/8

As I mentioned before, you will probably want to adjust and tweak the values for Kp, Ki and Kd after the calculations, as they might not be completely perfect for you scenario. Use the rules for finding the constant terms to aid you.

Extras:

The while loop:

You may have noticed that in all my pseudocode examples I have placed the main code within a while loop. This is because the error, integral, derivative and speed needs to be recalculated as the error changes. For example, if you made it calculate the speed once, but not again, it wouldn't be able to refresh and change the speed accordingly – it would just continue at its original speed!

The loop is vital to the PID controller – don't forget to add one!

So how do you tell it to finally exit the loop, once it has done it's job? Well, one of the common ways is that if you know approximately how long it takes for it to complete, you can make the loop run for a designated amount of time (but obviously slightly more than it needs – to be sure it does actually complete the loop).

Another way is to detect once the error has reached zero and has finished. If you choose to go this way, be careful to make sure it has detected that it has completely stopped. For example, if you told it to run the loop *until the error reaches 0*, if there is any overshoot, nothing will be able to be done since it will have stopped (for it to overshoot, the error must have passed a point of 0). So, you could get the loop to see how long the error stays at 0. If it is only at 0 for a very quick amount of time, chances are that it has overshooted and will need to re-correct itself. Or, if it has stayed at a value of 0 for a longer time now, it will be safe to say it has come to rest.

Resetting the integral and previous error:

So I have already covered that sometimes you need to reset the integral to 0, but there is one last time it needs to be at a value of 0. When you begin the loop, the code is automatically assuming that the integral is at 0 and that the previous error is what it should be. However, if the loop has been run before, the value for the integral and the previous error will still be what it used to be.

This can be fixed by setting the values for the integral and the previous error to 0 just before it begins the loop.

Summary:

Proportional – your error, the value between where it is now and where it should be.

error = (target value) – (sensor reading)

Integral – the running sum of previous errors, used for making fine movements when the error is small.

*integral = integral + error*dT*

Derivative – the change in errors, used to predict what the next error might be.

*derivative = (**(current error) – (previous error)**) / dT*

The loop – the calculations all need to be run inside a loop – don't forget to include it!

Put the three components together, plus some accurate values for Kp, Ki and Kd, and you will have a very consistent and accurate controller.

I hope you have enjoyed this guide, and have understood it completely and learnt what you can from it. Look out for my other programming guides, currently also available is the Beginners Guide to ROBOTC.

Written by George Gillard,

Free Range Robotics, VRC #2921,

http://robotics.org.nz

Thank you to all of those who have helped to make sure this programming guide is as accurate, faultless and helpful as possible.

Last edited: 10 May, 2012